THE BUCHAREST UNIVERSITY OF ECONOMIC STUDIES

THE FACULTY OF ECONOMIC CYBERNETICS, STATISTICS AND INFORMATICS

# DISSERTATION THESIS

SCIENTIFIC COORDINATOR:

CRISTIAN – VALERIU TOMA, Ph.D.

GRADUATE:

BIŞAG ALEXANDRU - ŞTEFAN

BUCHAREST

2022

THE BUCHAREST UNIVERSITY OF ECONOMIC STUDIES

THE FACULTY OF ECONOMIC CYBERNETICS, STATISTICS AND INFORMATICS

# DISSERTATION THESIS

## DESIGNING A PRIVACY-ORIENTED SYSTEM FOR FILE-STORING AND FILE-SHARING

SCIENTIFIC COORDINATOR:

CRISTIAN – VALERIU TOMA, Ph.D.

GRADUATE:

BIŞAG ALEXANDRU - ŞTEFAN

BUCHAREST

2022

# Table of Contents

# Introduction

What a society perceives as a file has changed multiple times throughout history and, in a practical way, the existence of objects that encapsulate information might even predate the sense of perception itself.

In order to talk about files, we must talk first about writing.

Writing, which was and still is the main channel used to store and exchange data and information, was developed independently in three regions, in different periods of time, the first, and the only one that could be traced back to its inception without interruption being the cuneiform script.

What's interesting about the cuneiform script, which was invented around 3200 B.C in Sumer, the geographical region of Iraq, is the fact that its origins can be traced back even further, to a form of writing which did not involve writing at all, but tokens of clay of different shapes and sizes marked with specific symbols that were created to count goods. [1]

The tokens were used during exchanges and similar objects were found in multiple places around the globe, showing that, even for the other writing systems that were yet to emerge, the first step was the ability of the prehistoric people to work with abstractions.

The following important abstraction was the ability to represent complex concepts through minimal depictions, such as small drawings, which also had a correspondence with the spoken language: the writing was now being used not only to store data but also to exchange information about actions, events or other people.

During the following millennia, the alphabets that were developed by different cultures became more complex and more able to express subtle nuances, being used as a counterpart to speaking. The only drawback was that writing, unlike speaking, was not instantaneous, and it required both effort and knowledge. To overcome this, educated people in the higher classes of societies started to work as scribes, and the documents that were produced were accessible to few people, thus the incentive to learn reading and writing was close to non-existent for the masses.

This changed with the printing revolution, started in 1450 by Johannes Gutenberg in Europe, who invented the first commercial printing machine. Before this event, there were notable efforts made by the likes of Bi Sheng and Wang Zhen, in China, that used clay and wood as printing mediums, but, even though successful at the time, didn't spark the interest that the Gutenberg Press did. [2]

With this technological advancement, the files were now easy to replicate and distribute across large geographical zones. In the following centuries, the printing technology advanced and spread out all over the world. The book became an object that could be owned, could be personal, and, besides the book market, which became a real business, the printing technologies revolutionized all the other industries.

After this important milestone, which transformed the file concept into the most tangible form that we can think of, the paradigm shifted once again when the ability to use abstractions to represent complex concepts was once again proven to be the most important skill that humanity has mastered.

The most recent revolution, and arguably the one that impacted the most people, is something that we can still witness today, as the printed materials are continuously transformed into ones and zeroes and stored inside electronic machines that could fit into our pockets.

The Digital Revolution that started a few decades ago, in the second half of the 20th century, was also the one that marked the beginnings of the Information Age [3] which shifted the ways in which data was perceived.

Because of this, the challenges that accompany files have changed as well, the storing of large quantities of documents becoming, from a problem of logistics, one of cryptography and security, and the problem of scarcity, one of privacy and trust.

The ubiquity of digital files and the plethora of tools created to read, write, exchange and transfer them might make it look like the problems around the *file concept* were completely solved and everything that's left to do is to use them without thinking of ways to improve the whole system – *but this is not the case*.

For what is worth, the clay tokens might have been the perfect tools for the prehistoric salesman, but, fortunately, he didn't stop there.

The scope of this thesis is to highlight the most important problems that emerge when a digital file has to be stored and distributed, and to come up with a plan to avoid them without accepting any compromise.

The first chapter analyzes the current state of the market and pinpoints the main issues that should be fixed, along with a perspective on how they can be mitigated.

The second chapter formulates a generic design that can be followed to implement a privacy-first file-storing and file-sharing system, without making any assumptions about the technologies that should be used.

The third chapter presents a selection of tools and technologies that can be leveraged to implement the design that was previously outlined, with technical explanations that motivate the choices that were made.

The last chapter reviews the progress that has been done in the thesis, formulates a conclusion and revisits the steps that have to be followed in order to improve the applications of the future by ensuring that privacy is a default characteristic, not a premium feature.

# Chapter 1 - The state of the present

## 1.1 - Ownership and trust

It’s no wonder that today most of the intellectual assets ever produced are digitalized.

Even more interesting, the technological processes around files have become so intuitive and natural that many people might have real troubles trying to explain where their digital property is actually located.

Sure, there are tech-savvy people who have a proper plan to store, secure, and back up their data, but for the people who see the technology just as a tool, which is the biggest part of the population, there’s a mantra that is being repeated by the biggest IT companies in the world for a couple of years now: *you should not care – just use the cloud.*

Undeniably, the cloud revolution is truly remarkable and should be embraced by everybody as an important step to a better future, as it provides unrestricted access to technologies that, until recently, were available only to those who were either living in a well-developed country or were working for a tech giant.

The benefits outshine the disadvantages by a large margin, but there are still important topics that should be brought into discussion such as security, openness, trust, and the ethics of having a handful of companies to virtually own the data of more than a quarter of the planet’s population. [4]

*“But it’s in the cloud”* - some may say – “*we should not really care how the files are stored and secured”*, and that’s true, but the problem is that, in an ethical sense, this could not be further from the truth.

What could not be denied, though, is that this belief is not the fault of the general public, as technology is not a topic that can be easily researched without a proper background. It’s harder even for specialists to understand what’s happening when the visibility inside the inner workings of a piece of software is close to zero, as the most popular file storage providers offer closed-source solutions.

Additionally, following the principles defined by the economies of scale in a market which requires little to no raw materials, some IT companies became big enough to control specific segments and made it impossible for independent, smaller competitors to exist without the back-up of another tech giant, as it recently happened with Microsoft and Slack [5].

The good thing for the customer is that, with almost infinite resources, big tech companies can produce software which that flawlessly, but this is also the source of a lot of problems.

For example, the ecosystem term, which was intensively promoted by Apple in the last decade, defines a group of software and hardware components which integrate so tightly that the user has a sense of continuity when using all of them at once. From a technology perspective, this is a great achievement, but, besides the monopoly allegations and the legal processes [6], the users of these products will have little to no motivation to leave the ecosystem, and this makes them less inclined to switch to solutions which are better for them from other points of view, such as *security and privacy*.

This does not mean that the biggest file storage providers offer insecure solutions: this would be, of course, false, as the best people in tech are part of these companies, but to discuss security a clear target must be decided first: *from whom should a file be protected?*

This question, which transcends security and blends into the realm of privacy, trust, and the ownership of data can be reassembled as: *who owns a good that can be infinitely multiplied and can be accessed by more than one party?*

And this is a perspective that, unfortunately, does not receive the attention it deserves.

## 1.2 - The state of the market

The privacy of the end-users is, most of the time, at the hand of the software development companies that benefit from their data to increase their profits and extend their operations.

The biggest issue with this approach, besides the ethical considerations and the famous precedents of client data abuse, is that, in the long run, the lack of privacy will become the de facto standard for new applications that are being built.

Fortunately, in the last decade, fueled by the increasing power that big tech companies have over their users, the subject of data privacy became very disputed, both by specialists and by enthusiasts.

This problem started to be addressed by federal actors as well, a few years ago the European Union adopting the General Data Protection Regulation (GDPR) to help its residents regain control over the data they are sharing.

Sadly, even though most companies became compliant, and the situation became a little bit better, at least for the people residing in the EU, privacy is still a hot topic, being intensely debated, but without a clear proposition.

Among billion dollars scandals, such as Facebook's affair with Cambridge Analytica, many independent projects were kicked off, marching on the idea of privacy. While the existence of more ethical alternatives looks more than appealing, the general user is yet to be convinced, as big tech companies' popularity on the market still increases.

Currently, the biggest cloud storage solutions destined for the general public are Google Drive, Apple's iCloud, Microsoft OneDrive, and Dropbox, none of which are open source. In terms of features, they offer cloud-based storage capabilities, targeting both enterprise and individual users, and are highly integrated with the ecosystems created by their parent companies, luring the users in on the promise of usability.

Besides these options, which dominate the market share with an undeniable margin, there are a few known outliers, such as pCloud, Tresorit, and MEGA, which offer some form of zero-knowledge end-to-end encryption, but their adoption among the public is very low.

Another worth mentioning category dedicated especially to technical users is composed of open-source file-storing solutions, like Nextcloud and ownCloud, which are built on the promise of off-the-shelf privacy and security, and targets on-premises deployments to keep the data on a private server, closer to its owner. Even though it features server-side encryption to protect files from unwanted access, Nextcloud doesn't enforce default end-to-end file encryption, leaving users who are not interested in the security details behind their preferred software to be in the wrong and believe that their files are accessible only to themselves.

With everything explained like this, it might look like the gap between the first category and the others can be closed with ease, but this is not the case. Without proper statistics being published by the companies regarding their real number of users and because of the tight integration that exists between the applications developed by the same company, analysts were able to estimate that Google Drive, the leading platform at the time, exceeded the threshold of one billion users [7].

## 1.3 – The missing piece

To conclude that the users do not have viable alternatives would be highly inaccurate, as dozens of competitors are already out on the market, thus the reason for not switching over to privacy-focused solutions is not related to the scarcity of good and available software.

One important reason could be that most of the users decide to rely on the cloud storage provided by the brands that also power their devices, meaning that, for the companies which are not tech giants and do not own a popular operating system, there's no real solution besides targeting niche markets, even if they ship a superior product.

While this might be true, the real core reason might also be the lack of discussion about real privacy and the lack of security specialists willing to go public and tackle the views of the tech giants. This leaves the regular user not only clueless about the true ownership of his files but also uneducated on a topic that will become of capital importance as more and more processes become digitalized.

However, there's one special category of users who, sometimes, become more than that and are, in general, highly susceptible to adhere to the march for better standards: the software developers, the specialists of the domain who have the power to start a pro-privacy revolution.

This thing was already observed by many companies and the software built specifically for software developers tends, in general, to have an open-source variant, which must implement security and privacy-related good practices, as the code can be analyzed by anybody. The open-source model gives the developers the chance to become more than just users by allowing them to contribute and improve a project by themselves.

This way, a solution can grow and reach the general public as a better, alternative product which is not exclusively owned by a company, and which can be periodically checked by third-party developers to ensure that everything is in check. In this manner, privacy-focused applications can gain more traction, becoming first-choice candidates for a category of users who can promote into the general market.

Building apps focused on user privacy is a must in a future where big amounts of data will become one of the most valuable assets.

On the other hand, this approach has already been implemented, as there are a lot of open-source applications that respect the user's privacy, so an actual implementation is not what the market is lacking.

*From my perspective*, the missing piece is a clear path that developers can follow in order to create more secure applications, a detailed example to make it obvious that privacy does not have to be sacrificed on the way and that applications can be as intuitive even with default, strong security practices.

To ensure that the change will happen in time, the programmers must be educated first, because they are the ones who could and should educate the non-technical users about the topics that truly matter – not the companies.

To sum up all the points enumerated in this chapter, I outlined a minimum list of requirements that define a blueprint for a secure, yet modern file-storing and sharing system, along with technology recommendations and how the implementation of such a system could look like.

# Chapter 2 - An overview of a privacy-focused architecture

This chapter presents the blueprint of a secure, scalable, and modern file-storing and sharing application in a technology-agnostic manner.

Each of the following subchapters, starting from the subchapter 2.2, will cover the design choices and technical details of a particular subsystem. The end goal is to be able to implement a system that adheres to the core principles declared in the subchapter 2.1, regardless of the platform chosen for the implementation.

## 2.1 – The mandatory principles

In order to ensure that a system achieves all the required goals, a couple of principles must be stated. They are mandatory and interconnected to some extent, so the omission of one of them might have an impact on the others.

1. **Zero knowledge – End-to-end encryption**

As concluded in the previous chapter, it's difficult to define who's the owner of an inherently replicable good when multiple parties have access to it.  There are solutions, like blockchain, but they only answer the question of ownership, not the one of privacy which has, at least, the same importance.

To adhere to this principle, the system should have zero knowledge of the content that it operates with and, similarly to how UNIX-based systems treat files, should treat everything as a collection of bytes, regardless of the original content.

Thus, the system should offer an end-to-end encryption schema that guarantees to the end-user that he is the only one who has access to the files stored inside the system unless he explicitly grants access to other users.

In addition to file encryption, all the metadata which are not directly needed by the system and can be used to analyze the behavior of a user should be encrypted as well.

Any system that has access to all the private keys involved in the cryptography process does not achieve end-to-end encryption and violates the privacy of the user, even if it does not read or modify any of the files.

2. **Responsiveness**

The system should be always responsive and inform the end-user about its state.

During file transactions, the user should always be informed about the completion percentage and should also be informed about any operations that involve him, such as receiving access to another file, or a file being updated by another user.

Additionally, the interaction between the user and the system should happen as fast as possible, and the tasks which can be done without the presence of a direct connection should be handled by the system without requiring the user to be connected.

3. **High availability**

The system should have a high availability percentage, otherwise users will be deprived of the right to access their own property.

If an increase in the number of requests is detected and the system's ability to handle all of them is endangered, it should automatically scale in order to satisfy all the incoming requests.

During the period when the service is operational, the system should have an availability percentage of, at least, three nines (99.9%)

4. **Fault tolerance**

If an internal fault that can be overseen happens while the system is running, the system should be able to overcome it, either by fixing the core issue or by adapting its behavior.

The faults should be as invisible as possible to the end-user, but the user should still be informed whenever a blocking situation occurs.

When an error happens in one of the internal components, the self-healing mechanism of that component should not affect the performance and the responsiveness of the others.

5. **Traceability**

The system should log each interaction between a user and itself, exposing these logs to the end-user at will.

No operation should be left unrecorded.

If a fault occurs, the administrator of the system should have access to the system logs up to the point where the system became unresponsive. The system might provide a unified interface where the operator has access to all the system logs.

When multiple internal services are involved in a transaction, the transaction progress should be traceable at the level of each component.

6. **Platform agnosticism**

The system should be built using technologies available on any platform or device that might be relevant for the targeted use case.

The deployment process should be similar on all the targeted platforms and should work the same in all environments, without requiring special configurations.

In the case of redeployment on a different platform, the system should not require code changes.

7. **Openness**

The system should be open source, the code being available to anyone who wants to inspect the implementation.

Additionally, the system should have an open and documented API, allowing other developers to use it in a seamless way, to build custom components and clients, or to integrate it into their own solutions. The system should be open, preferably, to all the relevant platforms and devices, the API being exposed through a standardized protocol, available everywhere.

The system should not be a final, enclosed application, but a component that can be plugged into other solutions to offer secure file storing and sharing.

8. **Loose coupling**

The system should be loosely coupled both internally and externally.

The *internal loose coupling* is applicable when the system is composed of multiple interconnected services. A service should be aware of the interfaces exposed by the other services but should not be aware of their inner workings. Besides, each service should have access only to its own data.

The *external loose coupling* refers to the methods in which the system exchanges data with external systems. The system should use abstract interfaces that can be replaced at will, the interface of the external service being known only to the component that establishes the connection.

9. **Access federation**

Any instance of the system should be able to communicate, at will, with other instances of the system, making possible the exchange of files between users registered on different instances.

The federated access should be available at the will of the user, but should also be controllable on a system level, in order to reduce the interaction with external instances, which, depending on the use case, might not be desired.

## 2.2 – The core

To achieve all the principles described in the previous subchapter, the blueprint of the server-side system, which represents the core component, will feature three subsections, the first two addressing the internal design, adapted to the targeted environment, while the latter presents the data entities.

### *2.2.1 – Thinking at scale – The Global deployment model*

The first design of the core system is based on a microservices architecture [8], a choice that is omnipresent in today's tech landscape and targets those use-cases where the system needs to handle thousands of requests per second and the available resources are virtually unlimited.

Ten years ago, one of the hardest tasks for programmers developing popular web applications was handling the large number of requests generated by the users. Back then, when the term of microservices was yet to be verbalized, other approaches were used to build load-resistant applications.

Enterprise models, such as service-oriented architecture, resemble, more or less, the key concepts that made the microservices skyrocket in popularity, but failed to deliver on the promise of simplicity: instead of a simple distributed system, developers had now to take care of a service bus as well, and they also had to make sure, mostly by hand, that everything is running as expected.

This does not mean that microservices are easy to implement and maintain, because that would be an understatement, but the technologies and methodologies of the present make the task of deploying and monitoring microservices a lot less tedious.

The idea behind choosing a microservice architecture is the horizontal scaling ability which, coupled with technologies that confer on-demand elasticity and high availability, ensures that the system will always be ready to take on new requests and respond without having to deal with resource-related problems. Besides that, microservices allow each independent unit to be developed with a different set of tools and practices, making the project easier to develop when multiple programmers are involved.

In the pre-containers world this would mean chaos, but, when the deployment process can be automatized regardless of how many different technologies are involved, this challenge, otherwise a difficult one, is almost trivial.

More so, with microservices, the concept of data ownership can be leveraged, each component having its own database that is exposed to the other services through a well-documented REST API, establishing a loose coupling between the actors.

The Global deployment model is built from seven different microservices, each one having its own particularities and responsibilities that, when combined, deliver a fully working system that can adapt on the fly.

The order in which the microservices are presented is not related to their relevance, but it will help to explain easier the interactions between them, the data flow, and how everything manages to come together.

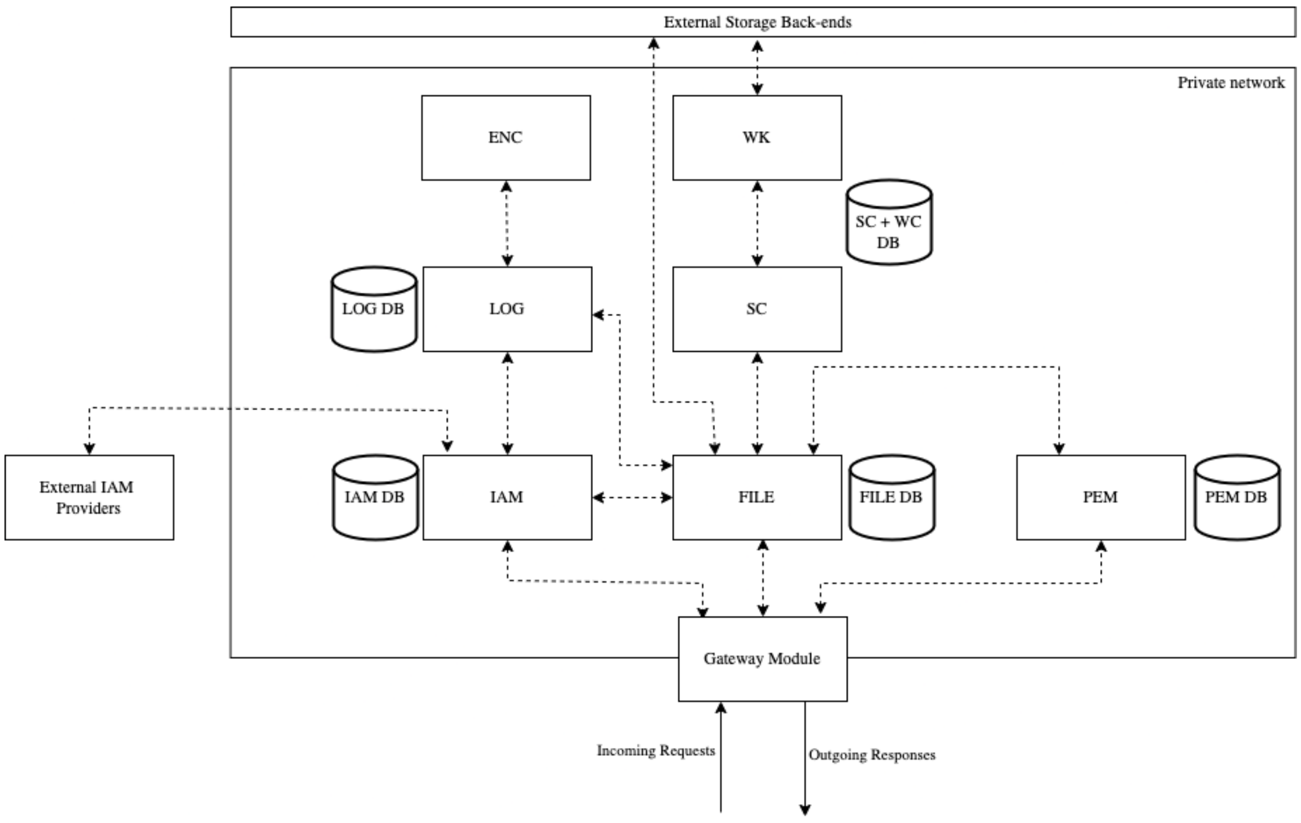


Figure 1 - The microservice-based architecture

1. *The Identity and Access Management module (IAM)*

The Identity and Access Management module stores information about each registered user.

Depending on how the system is used, standalone or as an abstract storage back-end for another application, the registering process might involve different steps, but the expected result is the same: a new user should be identified by a pair of credentials, such as a username and a strong password, metadata and, the most important detail, the asymmetric public key received as part of the registration request.

Optionally, if the system is integrated with another application, it can be configured to rely on an external IAM provider, ensuring that advanced features, such as single sign-on, are available and that the communication between the client application and the system is seamless.

Besides registration, the module will handle the 2-step login procedure that checks if a client who tries to authenticate knows both the credentials and has the right asymmetric private key.

The module will be queried by the other services each time additional information about a user is needed and will also be queried by the client when a file is shared or when the keys used to encrypt a particular file are refreshed.

Additionally, when federated access is enabled, the share, upload, and download processes will involve this service that has to provide information about the remote address of the federated server.

If required, the IAM module can be used by the system itself to prove its identity to the other connected instances.

2. *The Encryptor module (ENC)*

The Encryptor module has only one role: to encrypt, upon request, the information that does not represent a secret for the system, but which, in the long term, in case of a breach, can threaten the privacy of the users.

When the Encryptor module receives a request that contains a message, it will communicate with the IAM module and retrieve the asymmetric public key of the user.

Next, it will generate a random symmetric key and a random nonce and will use them to encrypt the received message. The symmetric key will then be encrypted with the user's public key and will be stored, along with the encrypted message, in the database.

The generated key should be destroyed immediately, and the memory should be overwritten.

An important notice is that the Encryptor module *should not be used to encrypt the user's files or any other type of information which should not be disclosed to the system.* Its sole purpose is to centralize the encryption process for the information which does not represent a secret for the system!

### 3. *The Logging module (LOG)*

The most important role of the logging module is to report each action that happens inside the system and store it.

The logs should be divided into at least 2 major types: *user logs* and *system logs,* each type being persisted in a different database instance.

The *user logs* will be written by each microservice when a specific action is taken by a user. Each log will be generated by one of the other modules after the processing invoked by the user is done.

The current action is known to the system because the system has to handle it, but, in the long term, the sum of actions taken by a user might represent information, instead of mere data, so encryption should be involved in this step as well, but, because the system is also aware of the information that will be hidden, the encryption will be done on server-side to reduce the number of interactions with the client.

Once the log message is received, the Logging module will pass the received payload to the ENC module and will receive the encrypted log and the associated encrypted key that will be stored in the database. Because the asymmetric private key is known just by the user, he will be the only one able to decrypt the logs once the information is persisted.

Upon request, a user has access to its own log history, each entry being received as a pair composed of the encrypted message and the associated encrypted key.

The second type of logs, *the system logs*, are written automatically by the system to trace requests, internal states, possible errors, and any other information that might be useful for the administrator to understand how the internal behavior.

Because these types of logs contain information about the current instance of the system, encryption is optional, as the internal processes are not a secret. The database in which the system logs are stored can be integrated with other search systems, outside of the system's boundary, to facilitate viewing, querying, and analyzing the recent events when the system becomes unstable.

### 4. *The Permissions module (PEM)*

The Permissions module defines user permissions for each file, and it's used by all the other microservices that need to find out the level of access that a user has on a particular file.

The access type could be either direct, by defining a group of fixed, individual permissions on each file entry and populating the values accordingly, or indirect, by defining different roles with different access levels and associating a role with a particular file.

The approach is not important from a security perspective, but, from a usability perspective, the service should be as customizable as possible, thus, the role-based approach might be a better solution in the long run, especially if the system will be used by multiple applications that have different scopes.

The only mandatory operation for the Permissions module is to ensure that a user who gains admin-like access to a file cannot remove the permissions of the original owner. Any other type of action is acceptable, as it does not threaten the integrity of the data.

If enabled, the Permissions module, together with the IAM module, is involved in the federated access process, ensuring that the remote users who require access to a file have the right to access it.

### 5. *The Files module (FILE)*

The core microservice is the Files module which handles the files that are uploaded or downloaded and ensures their integrity as long as they are stored inside the system.

The most important restriction that has to be respected is the zero-knowledge principle, from the module's perspective each processed file being nothing more than a collection of bytes.

The main purpose of the service is to guarantee that the users can upload files, either in a streamed or a buffered mode, and that the files are correctly preserved, regardless of the storage back-end that is used. The same thing stands true for downloading, the module being responsible to connect to the storage back-end and execute all the operations that are required for the client to be able to access his files.

In terms of performance, the File module must be as responsive as possible by minimizing the impact that moving big quantities of data inwards and outwards can have. To achieve this, the module can delegate the actual upload of the file in the proper back-end storage to other modules, but has to be sure that proper notifications were sent, both to the user, who should be announced that the processing of the file will be handled in an asynchronous manner, and to the other components that are involved.

Additionally, the Files module can make use of alternative storage solutions, like network-accessible disks or distributed file systems, to ensure that the communication and the internal handling of the files are not blocked, no matter how many instances are deployed.

It integrates with IAM and the PEM modules in processes like uploading, downloading, sharing, removing access, or refreshing the keys.

The service has a central role in the federated access scenario, being able to act as a client for remote instances, and operate as an intelligent proxy for its clients, by retrieving remote files and handling them seamlessly.

One more important feature is that the service should permit the user to resume an upload that was interrupted by uncontrollable causes without requesting again the parts of the file that were already delivered. Optionally, it can provide a mechanism to handle the exchange of multiple file chunks at once and manage the reconstruction of the content once everything is received.

A general recommendation is to adopt the streaming of content whenever possible, to the detriment of buffering, to prevent events, such as memory overloading, that might affect the ongoing processes of other users.

### 6. *The Scheduling (SC) and the Worker modules (WK)*

The scheduling module was indirectly mentioned in the Files module description, being one of the components that should support the file transfers between the systems and the external storage back-ends.

The main task of these two modules is to help the Files microservice to be as responsive as possible, by coordinating all the operations that should be executed on a file once the upload finishes. For this distributed action to execute efficiently, the Files, the Scheduling, and the Worker modules should have access to common storage that houses files until the processing is done.

If enabled, the Scheduling module is notified each time a new file is created in the temporary storage location. Each notification contains information about the new file that has to be processed. The information is stored in a database that is shared between the Scheduling and the Worker instances. After the notification is processed, the Files module receives an immediate response, letting its client know that the system started the handling of the file.

To prevent the scenario in which the same file is picked up by multiple instances of the Scheduling module, the instance that receives a notification assigns the operation to itself.

If one of the instances becomes inoperative at some point, the system will restart it, while keeping its identifier in place, preventing the loss of already assigned operations. The health checking and restarting actions might seem time-consuming, but, because of the asynchronous processing mechanism, the client is not actively waiting, so, even if the operation will be a little slower, it will not have a direct impact on the user and will ensure that each operation submitted to the Scheduling module will be eventually handled.

After a new operation is created and assigned, the scheduling part comes into action.

The first scheduled job executes at fixed or variable intervals of time and retrieves from the database all the operations assigned to the current service that were not yet processed. Once a job is identified, the Scheduling module makes a request to a Worker instance and marks the job as started, recording the timestamp.

Once notified by the Scheduler, the Worker instance starts the processing, which could imply recomposing the file stored in the temporary location from multiple chunks, and uploads it into the chosen storage back-end.

This is the step when the system interacts for the last time with the file before storing it, so this could be the insertion point for additional operations, such as compression.

During the whole procedure, the Worker constantly updates, in parallel, the job entry that was stored in the database.

When the Worker module finishes the upload, it notifies the Scheduling module which removes the file from the temporary location and notifies the Files module about the completion of the job.

If a worker stops while processing, a second scheduled job executed by the Scheduling module will observe that the job was not recently updated. The job is marked as stalled and, if the next loop finds it in the same state, it will be canceled, recreated, and assigned to a different Worker.

A third scheduled job runs from time to time and picks up the recently canceled jobs. They are passed to available Worker instances that roll back the partial uploads.

### 7. *The Gateway (GW)*

To avoid exposing the whole infrastructure, a classic Gateway module should be introduced to intercept the request from the users and reroute them to the proper microservice, acting as a reverse proxy.

The Gateway module should be as simple as possible, as no additional logic, besides rerouting, is needed.

Optionally, the Gateway could act as an identity-aware proxy, and validate if the incoming requests are allowed to pass. This operation is executed as well by each microservice, but request filtering at this level will prevent unauthenticated connections to reach the system, easing the load on the internal components and preventing random denial of service attacks to pass through.

The recommendation is to use an existing solution and configure it accordingly.

These blueprints can be used to implement a system that checks out the principles that were previously mentioned, being a solution that, powered by a great suite of tools, could satisfy high amounts of users.

This does not mean that everybody should start deploying loads of instances into the cloud, as this solution is, frankly, *overkill* for almost all realistic use-cases.

Unless thousands of users are implied, relying on a microservices architecture is probably not the answer, because, even with all the advantages that are brought to the table, the challenges and the costs are at least as important.

The next subchapter builds upon the concepts, the actions, and the actors that were presented to come up with a more down-to-earth deployment model that could be beneficial to individual users, families, tech-savvy groups, or even small-sized businesses.

### *2.2.2 – A private cloud – The Home deployment model*

Relying on microservices has a lot of benefits, but it also adds a lot of complexity.

For most use-cases, a monolith is more than enough to fulfill the requirements of a user base, but it should be modular enough that scaling can be achieved, if needed, without a full-fledged replication.

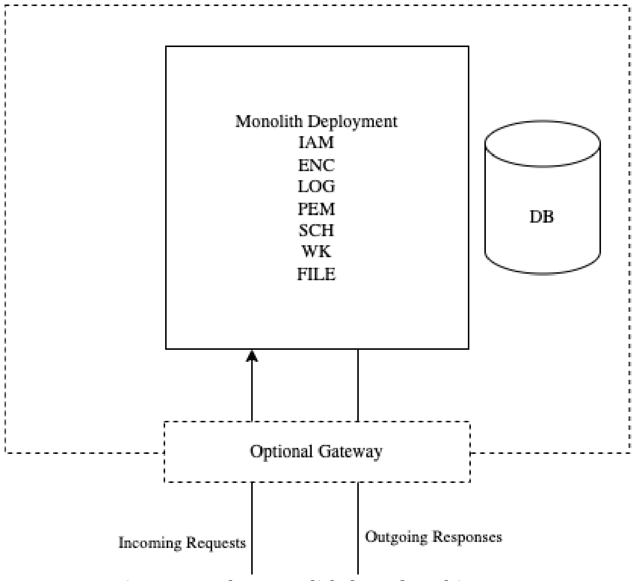


*Figure 2 - The monolith-based architecture*

The Home deployment mode encapsulates all the services described before as one but defines boundaries in the code so that, if needed, the internal "service" can be disabled and all the requests that should be processed by it are passed to one of the microservices defined above.

This model is traditionally called a modular monolith, but it goes a little bit further than this through the ability to horizontally scale independent components of the internal system.

For this to work, each join point between logically separate subsystems needs to be aware of the system status. The addresses of the services that are externalized should be provided during the initial deployment, as the system should also execute liveness probes from time to time.

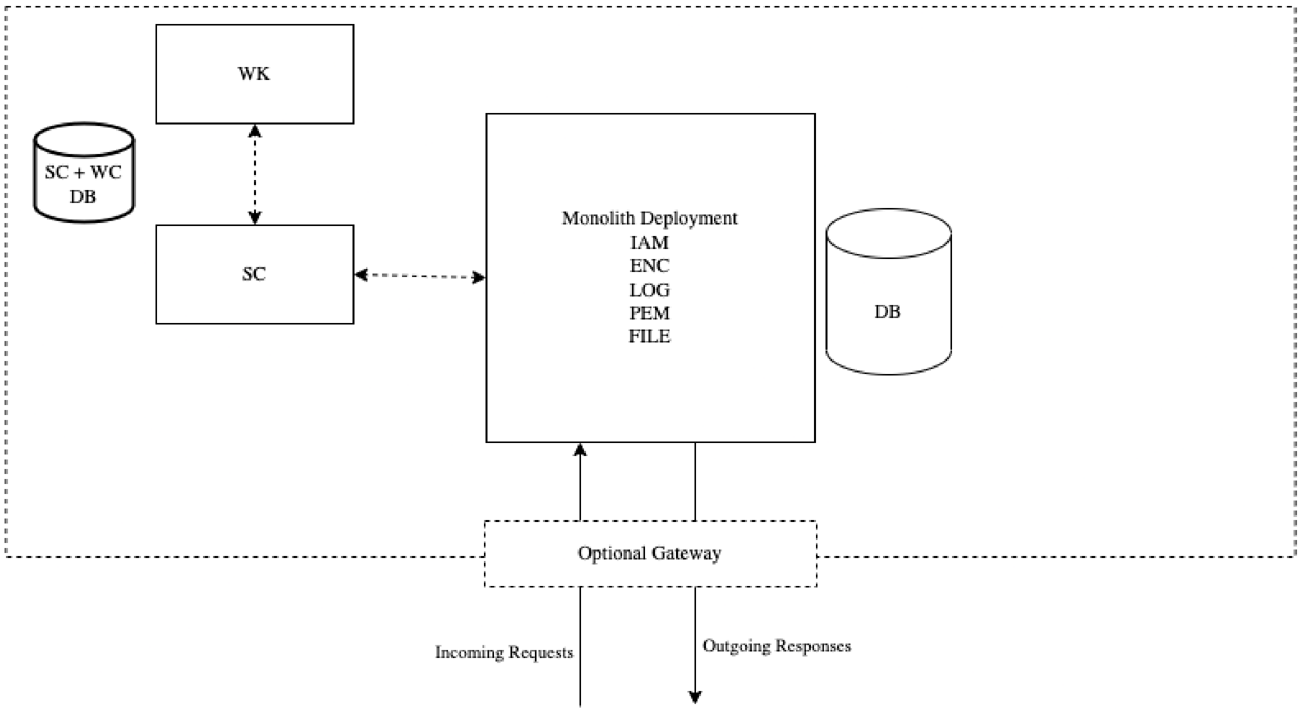


*Figure 4 - The modular monolith-based architecture*

Unlike the previous model, the Home deployment is a lot more careful with its resources as it will be, most likely, deployed on a single computer or a computer-like device.

Containerization should still be available to those who require it, but the system should be also published in its unmanaged format, letting the administrator adapt the deployment to their server and do fine-tuning.

To make this process as easy as possible, the system should come with a guided, CLI-based deployment guide. The administrator has to provide all the external resources that are required, such as databases, custom IAM providers, or remote file systems, but the deployment tool should handle the installation details. When configuring the system, the tool also takes into account the options of the user, and configures the database accordingly, creating only those entities that are not related to the externalized microservices.

To keep the system as light as possible, the gateway functionality is not integrated but has to be configured and added by the administrator, who has to make sure that the data exchange between the client and the internal components is not slowed down or blocked.

Compared to the cloud-native approach, where some technologies and platforms have already matured and should be preferred, the on-premises deployment should support a wider range of architectures, operating systems, and form factors.

For example, a valid use-case would be to deploy the monolithic version on a small single-board computer (SBC) that features an ARM-based CPU architecture, such as Raspberry Pi. Thus, the technical platform should be chosen to make the system as portable as possible, without making assumptions about the hardware that will be used.

In addition to that, depending on the use-case, some network-based components, such as the shared file storage between the Files, Scheduler, and Worker modules could be omitted, the monolith, compared to its microservices-based counterpart, being less reliant on network interactions.

With these changes applied, a system that supports both deployment models can be implemented at once, the components defined in the previous chapter being reusable to an extent, with only small, trivial changes being required.

The monolith approach should respect all the restrictions and recommendations defined for the previous model – the form factor change should not violate any of the core principles.

### *2.2.3 – Organizing data*

Each module should provide a way to insert, retrieve and query the metadata required to handle its tasks. Due to modularization, the data entities are simple, specific to each module, the data junctions being achieved through the interactions between the services, rather than through complex operations at the database level.

In the monolith scenario, the physical encapsulation of data, represented by each service having its own database, is not required, but the logical encapsulation should be preserved in order to facilitate the plug-off feature.

The schemas will be described as part of a relational model, but, in practice, non-relational databases can be used as well, as long as the relationship between the entities is preserved. Thus, the logical representation of the relationships, which is outlined using references, can be implemented with other mechanisms, such as embedded objects, if the technical differences do not affect the performance of the system or the integrity of the files.

The order in which the data structures are presented will mirror the order from the previous chapters, when possible.

## 1. *The Identity and Access Management database*

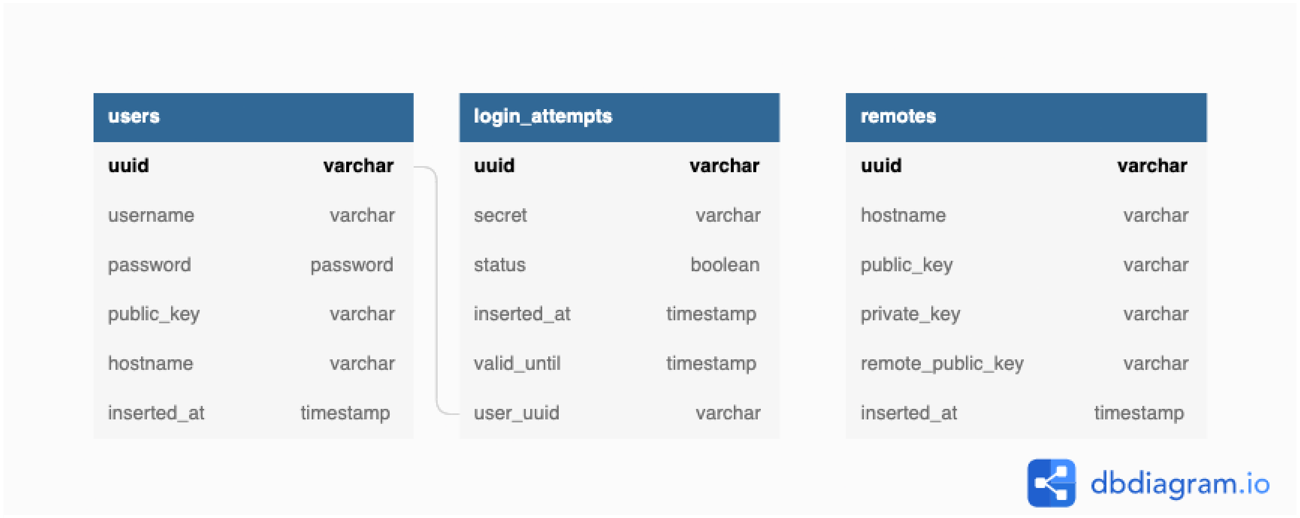


*Figure 5 - Identity and Access Management database*

The Identity database features three different entities, powering user-related activities, such as the login process, but also retaining system-level information used in federated exchanges.

## 2. *The Logging database(s)*

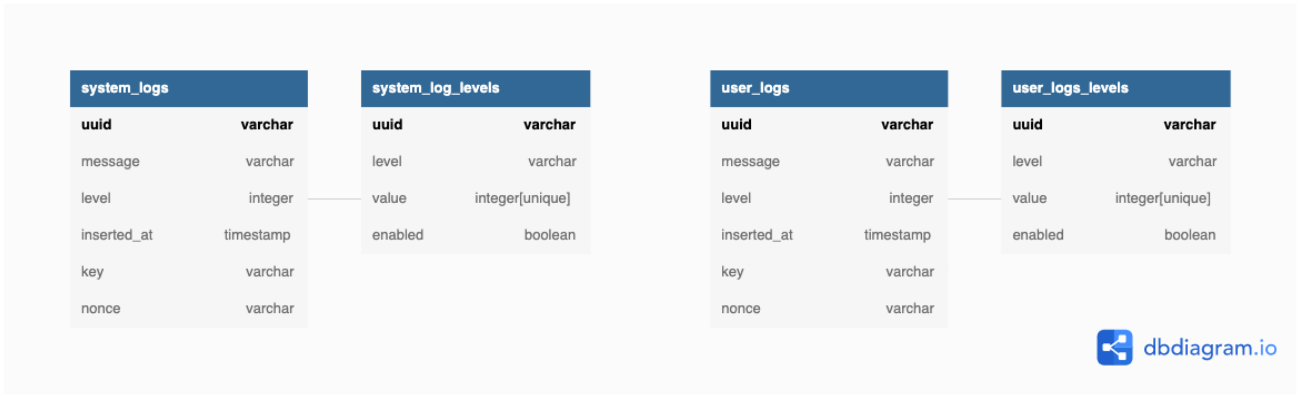


*Figure 6 - The Logging database*

The Logging database is composed of two similar groups of entities that are used to describe events two distinct categories of events.

The groups can be split into two databases, to make it easier for the administrator to pull the system logs into an external dashboard.

### 3. *The Scheduler and Worker database*

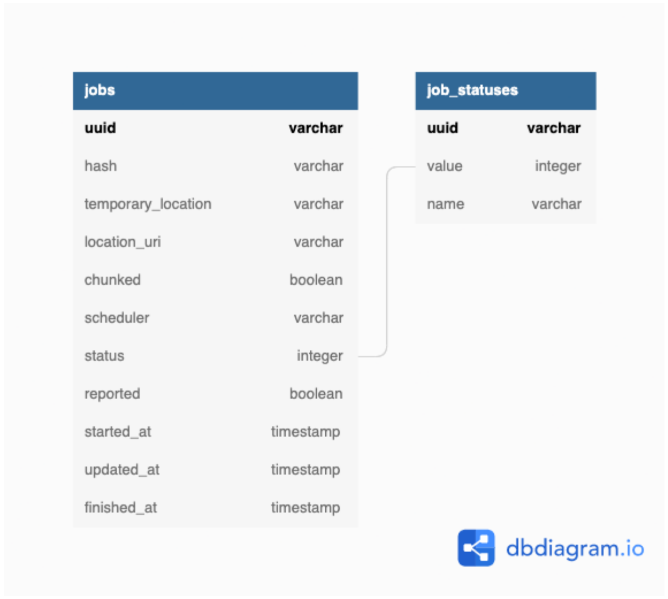


*Figure 7 - The Scheduler and Worker database*

The Scheduler and Worker database is the only one shared between multiple services, being used to track the progress of the asynchronous file processing mechanism.

### 4. *The Permissions database*

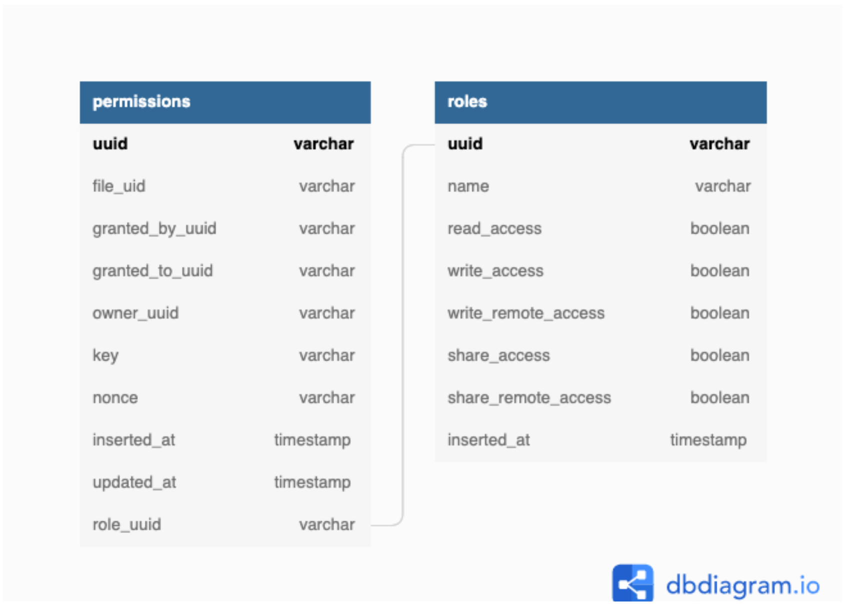
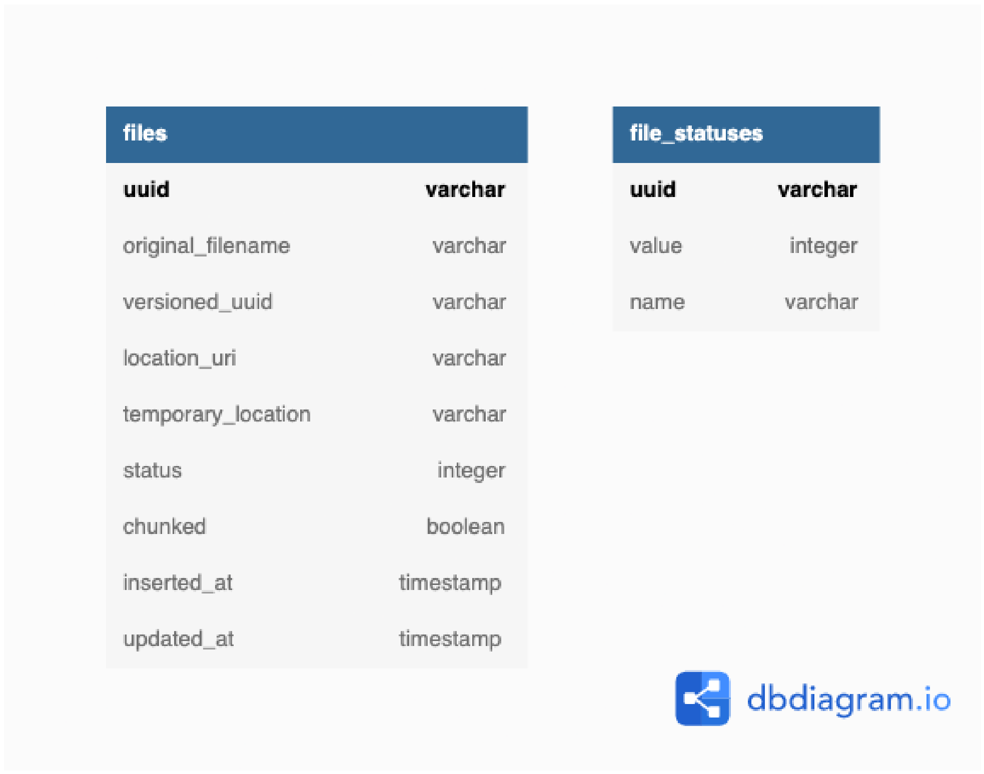


*Figure 8 - The Permissions database*

The Permission database is used to describe file-level permissions that can be granted and revoked by the users. It is one of the sections that can be customized, based on the user's needs, in order to achieve granular access control, through the roles mechanism.

*5. The Files database*



| files | |
|---|---|
| **uuid** | **varchar** |
| original_filename | varchar |
| versioned_uuid | varchar |
| location_uri | varchar |
| temporary_location | varchar |
| status | integer |
| chunked | boolean |
| inserted_at | timestamp |
| updated_at | timestamp |

| file_statuses | |
|---|---|
| **uuid** | **varchar** |
| value | integer |
| name | varchar |

*Figure 9 - The Files database*

The Files database stores metadata about the uploaded files, without being bound to an actual representation.

The scope of the table is to provide enough information to track the statuses of the files, access them without additional lookups and offer a versioning system capable of avoiding file system conflicts.

Unlike the featured examples, the Encryptor and the Gateway modules do not require a data storing mechanism as they are stateless and should execute an identical set of operations on all the requests, regardless of the input.

## 2.3 – The Client

The service is exposed, by default, through an intuitive API accessible for the applications that want to use it as back-end storage, but to make it easier for developers to interact with the system and ensure that the clients have the necessary tools to implement zero-knowledge content transfers, a client library should be created.

Because the system is generic and can be leveraged by multiple applications from different devices and technological platforms, it would be impossible to provide libraries for all of them. The first implementation should target the biggest potential platform and expose all its internal operations, in order to become a standard for the other libraries that, through open-source development, might be created by independent parties.

The first and the most important thing that the client library should do is to execute cryptographic operations before a file is uploaded and after it is downloaded. To achieve this, it should be able to use both symmetric and asymmetric algorithms to encrypt and decrypt the files and the symmetric keys associated with them.

The client should be able to generate random asymmetric keys and secure randomized values. The asymmetric keys are generated on the client-side when the user registers. The public key is sent to the server along with a password chosen by the user, which, together with the username, represents its credentials.

The asymmetric private key generated in the first step is vital, as the user would not be able to decrypt any of its files without it. So, after the user successfully registers, the private key is downloaded to the client's machine.

The problem is that, in order to ensure a safe redundancy, the client should multiply the key and store it in different locations.

Because of that, before downloading, the client library generates a random, secure master password and uses a symmetric algorithm to encrypt the private key. Now, the key can be stored anywhere because it cannot be used without the master password.

To upload and download files in a secure manner, the user must store the downloaded file and make sure that it cannot be lost, and, in addition to that, to remember the master password that was generated upon registration.

Without the file or the master password, the client does not have enough evidence to prove its identity and has no way to retrieve and decrypt the stored files. Because of this, the client library should inform the user to pay attention to these aspects.

When a user logs in, he should provide his credentials, along with the valid private key that is used to check his identity. If the login is successful, the user is able to connect to the system and retrieve his public asymmetric key.

For each uploaded file the client library generates a random, secure password and a unique nonce and makes use of a symmetric algorithm to achieve encryption. Besides that, it encrypts the password using the public asymmetric key of the user.

From a cryptographic perspective, the nonce must be random, but should not be treated as a secret, so it may or may not be encrypted together with the password. These file-specific credentials are stored by the service.

Without access to the asymmetric private key of the user, which is never shared with the system, the uploaded files cannot be decrypted.

The system should be able to handle the upload of large files, so, depending on the available options of the platform used to develop the library, different upload strategies can be leveraged, such as *the streaming upload* and *the chunked upload modes*.

1. *The streaming upload mode*

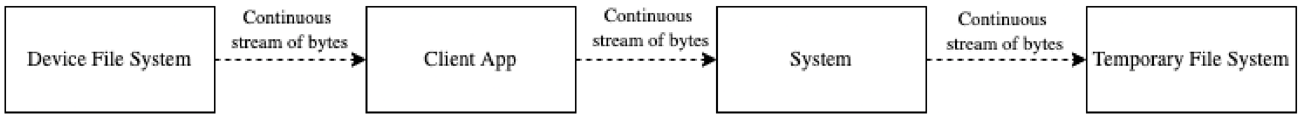

Device File System
Continuous stream of bytes
Client App
Continuous stream of bytes
System
Continuous stream of bytes
Temporary File System

*Figure 10 - The streaming upload mode*

The *streaming upload* mode is less popular because it's not supported by as many platforms as the chunked upload strategy, but it can make the whole system more responsive by reducing the impact of the file uploads on the client memory.

In a streamed approach, the file is read from the file system as a stream, and, depending on the chosen symmetric encryption algorithm, each bit can be encrypted right after it is loaded from the disk. In practice, though, more bytes will be read at once from the stream, so a block cipher algorithm could also work.

Once a bit is encrypted (more bytes in practice) it will be directly sent to the system as a continuous stream. Once a bit is sent, it can be dropped, thus reducing the impact on the user's device as the file is never fully stored in the volatile memory.

If, for some reason, the connection with the system is interrupted, the client can send a resume request and continue to upload the bytes that were not uploaded, without sending again the bytes that were already streamed prior to disconnection.

The reverse is true for downloading and decrypting files as well, the only difference being that, after decryption the file is streamed to the file system of the user's device, without being loaded at once into the random-access memory.

## 2. *The chunked upload mode*

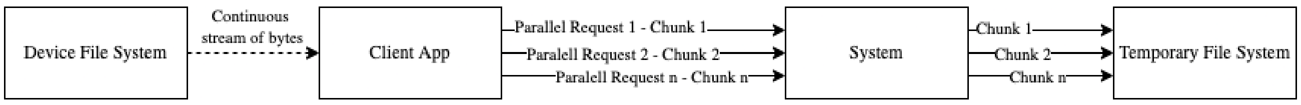

Device File System
Continuous stream of bytes
Client App
Parallel Request 1 - Chunk 1
Paralell Request 2 - Chunk 2
Paralell Request n - Chunk n
System
Chunk 1
Chunk 2
Chunk n
Temporary File System

*Figure 11 - The chunked upload mode*

The most used approach from a historical perspective is the *chunked upload strategy,* which is also known as buffering.

If supported, the loading and encryption part should stay the same, the only difference being that, instead of sending each encrypted "bit" right after it was encrypted, the client should store the result in memory and send it to the system in a more compact format.

If the file is small enough, chunking is not required, and the content is encrypted at once and sent to the system in a single transaction.

If the file is big enough, the encrypted content will be stored in a temporary buffer. When the quantity of data stored in the buffer reaches a certain threshold, an upload request will be sent to the server.

The client should not care how the server reconstructs the file, but it should provide, with each request, enough information to let the system know that it should process all the uploaded chunks as a big file, not as multiple, smaller ones, along with information about the ordering.

If streaming from the file system is not supported, the client library loads the whole file into the memory and divides it into multiple small-sized chunks.

Depending on the chosen encryption algorithm, the client might be able to process each chunk in parallel. After a chunk is encrypted, it is sent, as an encrypted chunk, to the system, along with the specific metadata.

In general, the preferred method should be the streaming upload, as it has a smaller impact on the client device resources, but the chunked upload strategy can be combined with it depending on dynamic factors and the level of support provided by the chosen platform.

It’s obvious that, with so many critical features, the client library is not just a tool used to access the system, but it’s an important part of the system as well.

Without a client that enforces all the operations mentioned above, the system, accessible through its API, is still usable as an abstract storage back-end, but the zero-knowledge principle is violated, as the content is not only known to the server, but it’s also stored in plain text.

The system is responsible for handling the files after the upload process ends, but the state of the content before the upload begins falls within the responsibility of the client library, as this is the only place that has access to all the information needed to execute cryptographic operations without compromising the user’s privacy.

## 2.4 – Security highlights

From a functional perspective, the security of the system represents the core feature.

To make sure that the files are protected and that nobody besides the user has access to them, modern cryptographic algorithms should be used.

The system uses both symmetric and asymmetric cryptography to achieve different goals. Symmetric-key algorithms, which are very performant and secure, are used to encrypt all the files and the metadata generated by the user with a unique key, while algorithms based on asymmetric keys, also known as public-key cryptography, are used to secure the secret symmetric keys, ensuring that only the holder of the private key is able to access them [9].

There are lots of algorithms, a lot of standards that are currently in use, and the cryptography world is in a perpetual change, driven by the threat represented by both faster, Von Neumann-like computers, which are still taking advantage of the Moore’s Law, and the quantum supremacy, which seems to be on the brink of inception [10].

The recommendation is to use standard algorithms which are mathematically proven and well-tested by the industry.

At the moment of writing, a choice that respects this condition is using *ChaCha20-Poly1305* for the symmetric operations and *RSA* for the asymmetric ones.

### *2.4.1 – Security & Integrity – ChaCha20-Poly1305*

ChaCha20-Poly1305 [11] is a popular cryptographic algorithm created by Daniel J. Bernstein in 2005 (ChaCha20) and 2008 (Poly1305) that’s being used in popular technologies such as TLS [12], WireGuard [13], and QUIC [14].

Besides confidentiality, conferred by most symmetric encryption algorithms, ChaCha20-Poly1305 is also an authenticated encryption mode, known as AEAD, which provides integrity by guaranteeing that nobody tampered with the cipher.

This is achieved by using ChaCha20, a stream cipher, that receives a 256-bit secret key and a non-secret 96-bit nonce as an input, based on which a keystream generator is able to produce unpredictable values. The generated bits are used to execute an exclusive OR (XOR) operation with each bit of the message that has to be encrypted.

Unlike AES, which uses complex mathematical operations, the key stream generator of ChaCha20 executes only three operations: XOR, add, and rotate, similar to an ARX cipher [15].
After executing the encryption, a cipher code is produced and the initial input is unreadable, the confidentiality of the sent message being achieved.

The second part of the name is represented by the Poly1305 cryptographic message authentication code (MAC) used to verify the integrity of the ciphertext, along with optional details called Authenticated data (AD in AEAD).

The cipher text previously produced (C) is glued together with the authenticated data, their paddings (a mechanism used to speed up the process), and their lengths using the following formula:

$$\boldsymbol{AD + pad(AD) + C + pad(C) + length(AD) + length(C)}$$

The result of this appending is used, along with a one-time key generated from the initial secret key and the nonce, as input for Poly1305. From this input an authentication tag is produced, which can be used to verify the integrity of the cipher during the decryption phase.

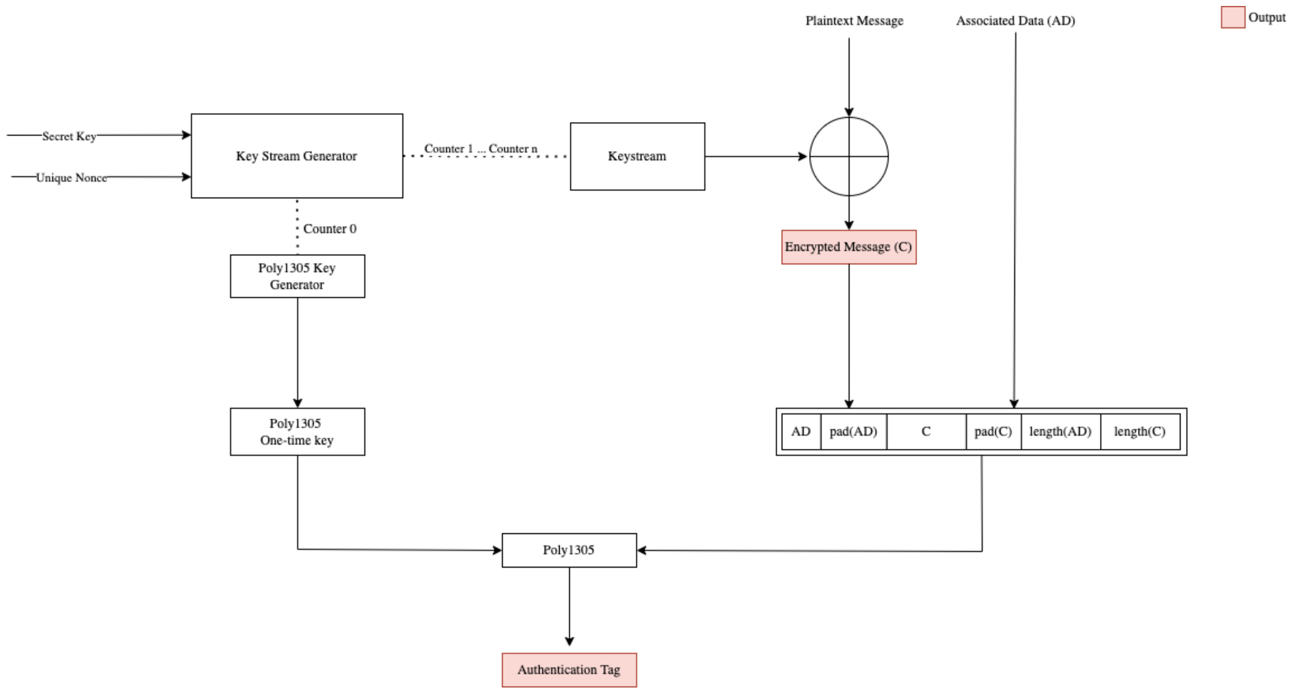


*Figure 12 - ChaCha20-Poly1305 [17]*

Another variant that uses the XChaCha20 algorithm is also popular, as it provides nonce misuse protection, but in the current system random nonces are generated for each uploaded file, so opting for XChaCha20, while better in general, does not bring many advantages.

ChaCha20-Poly1305 is a direct competitor for AES-GCM, which is also an AEAD and a more traditional choice, as AES is considered to be the current cryptographic standard, but the former offers better performance for software-only implementations, with AES-GCM being slightly faster when hardware acceleration is provided.

Despite performance considerations, both of them are good choices from a cryptographic perspective, being as secure as possible. If needed, ChaCha20-Poly1305 can be replaced with AES-GCM in the actual implementation if the impact of this choice is well understood.

However, because Poly1305 requires the whole encrypted message to be available at the moment of computing the authenticated tag, ChaCha20-Poly1305 cannot be used directly in a pure streaming approach, as data, which can be encrypted as it becomes available, cannot be verified in the same manner. Thus, the standard algorithm is better used in a buffered approach, where each chunk can be verified independently, the only requirement being that, during decryption, each chunk has to be treated independently by the system.

Fortunately, even though a streaming mode is not achievable with the standard implementation, there are ways to achieve streaming AEAD using segmentation of the plain text in a non-blocking, standardized way. Some of these techniques are already available in dedicated libraries [16] and can be used to solve this issue as well.

### *2.4.2 – Ownership – RSA*

Choosing RSA might come as a surprise, especially with the huge traction that elliptic curves cryptography gained in the past years, but for a system that has to directly encrypt and decrypt data, without having to do an actual key exchange, ECC cannot be used without additional operations which could slow down the whole system and could make the flows more complicated than they should be.

RSA, named after the three cryptographers who described it in 1977, one year after the Diffie-Hellman key exchange protocol was created, is the first public-key cryptosystem ever published that can be used for both encryption and digital signatures.

RSA is based on what is known as the "factoring problem" and assumes that, given two large prime numbers, is computationally hard to determine the factorization of their product.

The algorithm uses this principle to generate a public key and a private key as following, respecting the notations in the original paper [17]:

1. two large prime numbers are randomly chosen: p, q
2. the product of the numbers is computed: n = p * q
3. the Euler's totient function is computed: $\varphi(n) = (p - 1) * (q - 1)$
4. a number relatively prime to the Euler's totient function is randomly chosen: e
5. the modular multiplicative inverse is expressed as: d = $e^{-1}\ (mod(\varphi(n))$
6. the *public key* is composed of *n* and *e*
7. the *private key* is represented by *d*

The resulting public and private keys can be used to encrypt and decrypt messages. The advantage of public-key cryptography is that a message can be encrypted by either of the two keys and can be decrypted using the opposite one.

The mathematical process can be expressed as:

1. a message is encoded into a numeric format: m
2. the value of the cipher encrypted with the public key should respect the condition: $c\ \equiv m^e(mod\ n)$ which is solvable through modular exponentiation
3. the message can be retrieved using the private key by solving the equation: $c^d \equiv m\ (mod\ n)$

In the system, RSA will be used to encrypt the random symmetric keys that are used to encrypt the files. This way, the encrypted keys can be securely stored in the system, while only the user, holder of the RSA private key, is able to decrypt them and, by extension, to decrypt the files. The same principle stands true for any user-generated metadata encrypted through a combination of symmetric and asymmetric cryptography algorithms.

The disadvantage of RSA when compared to symmetric cryptography algorithms and even elliptic curves-based alternatives is its performance, being slower by a large margin. More so, the speed of RSA decreases if bigger keys are used and, according to NIST [18], keys which are, at least, *2048-bit* long must be used.

In the system, though, RSA is used to encrypt small-sized inputs, such as secret keys or encoded log messages, which means that, in practice, it will not affect the overall performance by much.

## 2.5 – Technical highlights

Similar to the previous chapter, where important details about the cryptographic algorithms that should be used were presented, this chapter features an in-depth analysis of the most important processes in the system.

A high level of detail is required to ensure that the design is understood and that, without taking into account the chosen technologies, an implementation of the system can be achieved without missing the relevant details.

### *2.5.1 - The Login process*

The Login process is designed as a two-step verification, inspired by the SSL/TLS handshake [12], which guarantees that the user trying to access the system has all the credentials necessary to prove its identity.

During the initial authentication step the client provides its credentials to the IAM module and, if they match, the microservice generates a random value and encrypts it using the asymmetric public key that’s stored in the database and passes that value back to the client. The random value is temporarily stored in the database as an authentication attempt that was not yet resolved.

The client receives the challenge and using its asymmetric private key decrypts the message. At this point, the client can prove that he knows which is the value of the secret value, but this evidence is not enough to prove that he decrypted the message and not just guessed the secret. Thus, using his private asymmetric key he encrypts back the message and sends it to the server.

The server receives the encrypted payload and, using the user's public key decrypts the message and compares it with the random value that was initially generated.

If the values match an authentication token is generated and returned to the client. The token is used by the client as part of the future requests to prove the completion of the login process.

If the values do not match the authentication is rejected.

Once generated, a token is refreshed while in use, but expires once the client stops from executing requests for a longer period of time. When a token expires, a new one can be generated by repeating the initial exchange.

### *2.5.2 The Upload process*

The Upload process is the most complex action supported by the system, the complexity being given by the multiple ways in which a file can be uploaded, as well as the faults that might appear because of the network.

After it loads a file from the client's device and encrypts it, the client library can upload it into the system in *buffered (chunked)* or *streamed* mode, as described in the client subchapter.

In the *buffered mode,* the file is uploaded in multiple chunks and stored independently by the File service in a temporary location. Before starting the upload, the client makes a create file request which returns a unique hash that represents the file in the system. In future requests, besides the hash and the file chunk, another parameter representing the order in which the chunks should be ordered has to be sent.

During this activity, the status of the file inside the system is *"uploading"*, this information being useful in case of network disruptions, as the system is aware that the file was not completely received. This mechanism is also used to prevent the re-upload of the already received chunks in case of network issues.

The system stores each chunk using the hash and the order parameter to prevent conflicts at the file system level. When all the chunks are sent, the client makes a completion request which lets the system know that all the required parts were uploaded. Then, using the generated hash and the temporary location, the File service will notify the Scheduler module and updates the status of the file to *"in progress".*

From this point on, the client's presence is not needed, as the processing of the file is handled by the system. The client can check the status of the file at any time, a log being created to let him know when the process is finished.

Once notified, the Scheduler module creates a new entry in its database and assigns it to itself. The first scheduled job that runs at this level will pick up the job and will offload it to a Worker instance.

The worker instance receives all the information needed to recreate the file, process it, execute any additional operations, and store it in the back-end storage of choice. Meanwhile, it is asynchronously updating the job entry in the database shared with the Scheduler module.

When all the steps are finished, it notifies the Scheduler which removes the file stored in the temporary location and calls the File service. The service receives the uniform resource locator needed to access the file in the external back-end storage and uploads the status to *"processed"*.

After this step, the client can access its file, update it, share it with other users, or remove it.

If a worker module encounters a problem during the processing of a file and becomes unresponsive, it will fail to update the job entry. The Scheduler makes use of a second loop to pick up all the jobs which were not updated for a specific period of time and marks them as stalled.

The job can exit this status only if the assigned worker resumes the processing. If this does not happen, the Scheduler cancels the job, recreates it, and assigns it to a different worker.

In the *stream mode,* the process is similar, but instead of uploading multiple chunks at once, the file is streamed by the client, bit by bit, into the File system, which stores it in a temporary location. If any networking issues occur at this point, the client can resume the upload using the previously generated hash and send the rest of the content.

When processing the file, the system treats it as a chunked exchange and recomposes it after the communication with the client ends. The rest of the flow is identical, and, because of this, the system can switch from buffering uploads to streaming uploads dynamically, in order to optimize the exchange.

The update of a file is executed as a regular upload, the only difference being that, instead of asking for a new unique file identifier, the existing one is reused.

If required, the File module can use an additional hash to represent the file content, letting the system store multiple versions for the same file without creating conflicts between them.

### *2.5.3 The Download process*

As expected, the Download process happens in reverse, but it's simpler, from a technical point of view, as it implies only the File module to access content that is stored in the external backend storage.

Once the user requests the download of a file, the client library communicates with the Permissions module to get the encrypted symmetric key. After it decrypts it using his private asymmetric key, which is not known by the service, it launches a download request.

The File module identifies the file, checks the permissions of the user, and starts to stream encrypted content from the external back-end storage to the client.

The client receives the encrypted bytes and decrypts them on the fly using the previously decrypted symmetric key. Once decrypted, the content of the file is streamed into a user-chosen location on the device's file system.

If any networking problems occur during the transfer, the client can resume the download process, using the current progress (in bytes) as an offset. The system treats this request as a new download, but instead of streaming the whole file, it asks the external storage to apply the offsetting as well.

### *2.5.4 The Sharing process*

The zero-knowledge policy that resides on end-to-end encrypted services makes the process of sharing content between multiple users seem more difficult compared to the regular scenario that's part of unencrypted systems, but this preconception is not necessarily true.

If a user wants to share a file with another, the client library retrieves the public key of that user and the encrypted symmetric key of the file. It decrypts the key using the owner's private asymmetric key, encrypts it back with the public asymmetric key retrieved from the server, and sends the result to the Permission module, along with the desired access levels. The system stores the new permission and creates the necessary logs to trace the action.

When the user who received access to the file wants to download it, it will simply retrieve the permission from the database, decrypt the symmetric key with its own private asymmetric key and start the download process. With enough privileges, the user is also able to share the file with others by repeating the same process.

The part which is more difficult is revoking the access, because, besides changes that happen in the Permission database, the key has to be changed to guarantee that, in case of a breach, the users that had access in the past do not have access anymore. Because of encryption, this operation requires a sustained interaction with the client, being equivalent to a combined download, file share, and upload process.

The permissions of the user who wants to revoke access of another are checked, and, if allowed, the client library downloads the current key of the file. After decrypting it, it generates a new random symmetric key and starts to download the file, decrypting it, on the fly, with the old key, and encrypting it back with the new one.

Depending on the preferred options, the file is either buffered or streamed into the file system, which updates the references of the existing entry to point to the new version. Then, the client downloads all the public keys of the users that should still have access and encrypts the new symmetric key with each of the public keys, obtaining multiple encrypted versions of the same symmetric key. It aligns the encrypted versions with the users and requests the update of the existing keys with the new ones.

As mentioned before, this process is time-consuming, and should be used with care, but, if the access removal is a common scenario for the targeted use case, the system can define a “key refresh” procedure that combines all the previous steps in a seamless way.

The advantage of this procedure is that users can be encouraged to update the keys from time to time, a common practice in all cryptography-based systems, to achieve an even better grade of security. This way, the steps behind access revocation can be reused for an additional scenario, motivating the high cost of implementing it.

### *2.5.5 The Server-to-server connectivity*

One of the key features of the system is its ability to be integrated with virtually any application that needs a secure and abstract storage back-end. This means that applications owner will likely use their own instances to make sure that the integration is as seamless as possible.

This aspect creates a big opportunity, as the system, regardless of its implementation, follows the same architecture and makes use of similar concepts and structures. From this perspective, two instances should be able to communicate with one another, achieving what is also known as federated access.

The bold idea, though, is not to connect two file-storing applications, which is also achieved by other systems, such as Nextcloud, but to connect different applications from various domains and allow them to exchange files in a seamless way.

For this feature to be achieved, both systems must be exposed to the Internet or to another type of shared network.

If this requirement is fulfilled, the administrators have to activate the federated access feature on their IAM modules. This will generate an asymmetric key pair and will exchange the public keys between the two servers. The keys will be used to verify the identity of the servers. Each time a remote call is requested, a server will encrypt the request with his own private key and with the recipient's public key.

The receiving server decrypts the message using the opposite key combination, ensuring, this way, that the identity of both systems is verified.

From the user's perspective, the connection is transparent, as the systems will communicate on each operation that requires remote exchanges.

To tackle the disadvantage of at-distance network calls, the information on the shared files, involving the IAM and the Permissions module, can be duplicated in real-time.

When, for example, a user grants access rights to one of its files to a federated user, the permission entry will be created in both systems, letting the remote user know that he received federated access to a file.

If a user requests access to a file, the Permissions module checks the "hostname" property and, if it resides on another instance of the system, it will support the Files module to connect. The Files module acts as a client for the paired system and streams the file directly to the client as soon as data bytes become available, without fully loading it into the memory.

This approach of server-to-server communication, though faster and convenient, opens each of the systems to a security risk that cannot be overlooked in case of a breach: if two systems are connected and one of them is compromised, the hackers will get access to the files stored in both systems.

The files are encrypted, so the hackers do not have direct access to them, meaning that the user's privacy and security are not compromised, but the threat is more relevant for the system administrators who should periodically check if the paired systems are still secured.

To eliminate this potential attack vector, the processes involving remote interactions should leverage one more step, executed by the client library.

In addition to the server-to-server authentication process, on each request, the client encrypts, using its own private asymmetric key the current timestamp and sends it, along with the unencrypted version. These fields are passed down to the remote system which has access to the asymmetric public key of the user and can verify that the request is authentic.

This way, operations that should be requested on a user's behalf cannot be executed without its presence, making an attacker unable to operate on the files stored on a remote server.

# Chapter 3 – A selection of tools

The previous chapter outlined all the design choices that should be taken to design a file storing and sharing system that’s secure, reliable, and scalable. The explanations were as platform-agnostic as possible to allow potential developers to use whatever they like as part of the implementation, as long as they comply with the mandatory principles.

The few restrictions that were mentioned do not reduce the technologies pool by much, as the core features are supported by multiple languages, frameworks, operating systems, and management platforms.

This chapter proposes a technology stack, but it should not be considered the only viable choice. I opted for these technologies due to my personal experience, so they should be considered *an implementation* of the previously detailed guidelines, but *not the implementation*: there might be other programming languages, databases, and frameworks that are even better suited to fulfill the requirements.

The one who implements the system should make a choice and compare the available options.

## 3.1 - Implementing the system

The first decision in terms of technology is made to ensure that the system has high availability and that it can automatically scale if the level of incoming requests increases: *Kubernetes*.

Deploying and monitoring microservice-based applications with Kubernetes is almost an automatic task, as the main thing that has to be provided is the description of each resource, expressed in YAML.

This approach, also known as infrastructure as code or IaC, can be improved even more with the usage of *Helm*, a package manager for Kubernetes configuration files, which helps in organizing individual deployments and services into flexible units called charts.

The first choice will, most likely, reveals the second one, as Kubernetes is used to orchestrate individual units called containers. To build containers, *Docker* is the de facto standard and it’s more than well suited to package microservices. Docker also helps in targeting multiple platforms at once, such as different processor architectures, as it can generate multiple images at once, each one being guaranteed to produce a container that runs the same, from an application perspective, on any type of system.

When it comes to the platform of choice, it's safe to say that the *Web platform* has the greatest traction and will probably continue to do so for a long time, as Web APIs can be integrated with almost any other platform, such as mobile operating systems or IoT devices. The system will expose a *REST*, or at least a *REST-like interface*, that can be accessed by clients through the *HTTPS* protocol, which guarantees that the exchange of information is private.

In terms of development technologies, the Web platform benefits from such popularity that virtually any programming language can be used. There are, of course, more mature technologies that had a great exposure to this kind of scenario, one of which being *Java*.

The first thing that describes Java as a programming language is versatility, as, from its inception in 1995, it was promoted under the WORA *(Write once, Run anywhere)* slogan.

Historically, Java was mostly used by enterprises to build complex systems, but it was quickly picked up by the programming communities around the world, one of the greatest perks of the language being the tools that were built around it.

In the last decade Java became almost synonymous with microservices, as both huge corporations and emerging businesses put a lot of effort into scaling their digital infrastructure. The most popular example is Netflix, which developed the now-famous Netflix OSS stack which was later integrated with arguably the most influential Java-based framework: *Spring*.

The Spring framework has a lot of components that can be used to build various types of applications. To implement the system, one of the best choices available is *Spring WebFlux*, a framework that can be used to build fully reactive applications. WebFlux is backed by *Project Reactor* [19], an implementation of the *Reactive Streams* specification [20], that defines an asynchronous core, implemented with the help of an event loop, the same model that is used in NodeJS as well.

Due to its asynchronous nature, reactive applications are well suited for IO-intensive scenarios, being able to handle more requests than a traditional, synchronous, servlet-based alternative.

For the application to work as expected, the external services that the system has to communicate with should support reactive interactions as well. At the moment of writing, reactive support for relational databases is not yet mature, even if there are notable implementations, such as R2DBC, so, as an alternative solution, one of the most reactive friendly options is *MongoDB*, a NoSQL, document-based database.

Besides the database, the system will interact, through the Worker module, with external storage back-ends. To comply with the reactive requirement, I settled for *GridFS*, a MongoDB system that is used to store large quantities of data, and the popular cloud-based storage, *S3,* offered by Amazon Web Services. Just like the system, these services will have access only to the encrypted version of the files, so security and privacy are not affected.

If the system administrator chooses to not use an external storage back-end, a viable option for instances that do not require too much scaling, an external, network file system can be mounted natively by Kubernetes and shared across the pods that operate with files.

In the same manner, when an external storage backend is used, an NFS can be integrated to ensure that the files, the scheduling, and the worker modules have access to a common writable location which will be used to temporarily store the files received from the client until they are processed.

Depending on the use case, the system might need to integrate with an external Identity Access and Management provider, and an open-source solution that is commonly used with Java-based applications is *Keycloak*, which can be integrated with the IAM microservice to grant access to the users using the *OAuth2 protocol* and *JWTs*.

If the management of the Keycloak instance proves to be too difficult, alternatives such as *Okta, GitHub Identity Platform, or Azure AD* are available and can be easily integrated instead.

The Encryptor module should be able to execute both symmetric and asymmetric cryptography, and the recommended algorithms are ChaCha20-Poly1305 for symmetric operations and RSA-2048 or RSA-4096 for asymmetric ones, as explained in the previous chapter.

With everything operational, the last step that needs to be done is to expose the services to the Internet.

As previously mentioned, the system itself does not feature a built-in Gateway component, so anything can be used. One of the most common choices is *Nginx*, a web server that can be used as a reverse proxy and can also be configured as a *Kubernetes Ingress Controller*.

The same selection of technologies can be used to develop the modular monolith that was described in the previous chapter as well, the only redundant aspect being Kubernetes which, if the system is used as a whole and no component is extracted as a microservice, does not confer any real benefits.

## 3.2 – Implementing the client library

The reasoning behind choosing the platform on which the client library could be done is similar: adoption rate and technological advancement.

From this perspective, there's no doubt that Web-native technologies, HTML, CSS, and JavaScript, have the largest user base. It seemed natural to select JavaScript (or TypeScript) as the reference language, given the fact that the browser APIs that are available, even though experimental, are very powerful in terms of file handling, connectivity, and streaming.

Besides the language itself, the most important aspect of the library is to not make any assumptions about the domain of the application that will use it and to not add any operations which are not specific to its tasks. Files will be treated the same, regardless of their content, and the operations executed upon them will be identical.

One of the most important things for the client library is to have access to the files on a user's device if the user allows it. Due to the *File System Access API*, this feature can be easily implemented, as JavaScript, once receiving the permission to access a particular file, can read it, write it, navigate through it and even truncate it.

Once access to a file has been granted, the next important step is to load its content into the browser's memory. This result, like the upload and download processes themselves, can be achieved through two different approaches, by streaming or by buffering the content.

The method that is easier on the client's device resources is streaming the content, which can be achieved with the help of the *Streams API* which can read data from the local file system sequentially, without storing everything in memory.

The same approach works in reverse as well, with the content that it's being downloaded being streamed directly into the file system.

After the content of the file begins to be loaded into the memory, the client library should encrypt the data as it becomes available. To do that, it must know how to implement symmetric and asymmetric cryptography operations, using the ChaCha20-Poly1305 and RSA algorithms.

Doing complex computations is a difficult task, and cryptography, which uses a lot of computations, is no exception. When working with a single-threaded language, like JavaScript, in a sandboxed environment, like the browser, doing intense computations is not easy, because blocking the main thread means blocking the application itself.

Fortunately, the browser offers two features that can improve the performance and avoid freezing the UI: *WebWorkers* and *WebAssembly*.

*WebWorkers* [21] are, more or less, the equivalent of multithreading for the Browser, allowing the main JavaScript thread to offload operations to other threads that execute in parallel, in the background. A WebWorker and the main thread can exchange data back and forth through messages controlled by programmable event handlers that resemble callbacks. Intense operations which would traditionally freeze a browser tab can now be executed in the background and communicate the results to the main thread.

With this addition, freezing is no longer a problem, but JavaScript is a high-level language that aims to offer developers useful abstractions. This means that, compared to lower-level languages, intense computations executed in JavaScript will be slower.

Thus, encryption and decryption are not blocking the main application anymore, but, because the computations might take a while to complete, they "block" the user by taking more of its time to finish file exchange operations. This is where *WebAssembly* can help.

WebAssembly [22], or WASM, is a technology that allows the virtual machine of the browser to run an additional low-level language along with JavaScript. The language is actually a binary format that is not meant to be written by hand. Instead, special programs can convert instructions written in other programming languages to WASM.

The number of available languages is continuously growing, but the most important thing for the current use case is that it allows low-level languages, such as C, C++, and Rust to be used for computational-intense operations – such as cryptography.

Because WASM is a revolution for the browsers, the technology has a lot of support from the community, with various WASM algorithms being already available, including ChaCha20-Poly1305 and RSA.

Combining these two techniques, the library will span a WebWorker that will execute cryptographic operations more efficiently with the help of WebAssembly.

After the content that was streamed is encrypted, it will be sent directly to the system, using the streaming upload mechanism provided by the fetch module. This feature is experimental and it's currently under debate, so the support in browsers is very limited. Until there's a consensus about it, the streaming upload mechanism can be replaced with the buffering approach that was described in the previous chapters.

The downloading process operates in reverse, and it involves streaming encrypted content from the system to the client, where it is decrypted by the library and streamed into the file system, into a file selected by the user. Unlike streaming uploads, streaming downloads are supported by all the major browsers and servers, being very useful especially when a client interacts with a reactive system, due to the existence of the backpressure mechanism.

With these technologies, the architecture presented in the second chapter can be implemented without compromising any of the mandatory principles that were listed. The choices are backed by my personal experience, so they’re not a recommendation, but just an example of how the implementation can be achieved and how it might look from a practical point of view.

Any other platform, programming language, or framework can be used as long as the original design of the system is respected.

# Chapter 4 – Conclusions: a future of privacy

As highlighted in the beginning, the slow adoption rate of security and privacy-focused apps by the masses cannot be attributed to the lack of options, but there are other reasons, not strictly technical, that make the process difficult.

From a systematic point of view, relying on the individual actions taken by each independent actor is a flaw, thus, the current approach of hoping that one day the users will become more interested in technical topics is wrong.

*The solution must be part of the system, not a spontaneous event that might or might not occur*.

The real change is to create discussion around these sensitive topics and educate the people that can produce an impact.

The change, *from my perspective*, should come from software developers, the people who can understand the risks of a privacy-deprived future, who have the influence, as experts, to guide the general public to more ethical alternatives, who know how to contribute and to improve digital solutions and, in the end, the people who can put direct pressure on the companies.

In addition to that, developers should not take for granted the positive phenomena that are happening in their industry and should always push to keep them relevant, because, besides their positive effects, they are also proof that change can happen.

For example, the open-source development model shifted the way in which tools, especially developer tools, are built, and encouraged (or forced) multiple companies to open their codebases or be left behind.

In the beginning, the open-source seemed a utopic idea that was rejected by the top tech companies, Microsoft being the most remarkable example [23]. With enough push, people that were once students or programmers at the beginning of their careers shifted the perspective once they started to work for and run these top tech companies.

Even if user privacy might not seem as critical as open-source, it is a subject that, if ignored, will slowly transition into obscurity.

This is a big threat, not because of how systems are built today, but because of them becoming de facto standards and influencing the systems that are yet to be built. This creates a huge technical debt that would be harder to mitigate once more and more software is created and the levels of abstraction increase.

Naturally, it seems easier to postpone the difficult actions that must be taken, but, if enough time passes by, it might be too late or too hard to fix the damage that has been done.

The main point of the paper is not so much to build a universal solution that must be adopted by everyone right away, because this would be impossible.

The scope is to demonstrate that security and privacy do not have to come at the cost of usability.

Designing a system that is both secure and oriented toward the user's privacy is not an easy task and, even though the outcome of the current article represents a *good enough* framework, a complex, failproof system is not a feat that can be accomplished by a single person.

In addition to that, while I was preparing the diagrams, selecting the tech stack, and trying to understand the core issues that make the general public to rely more on blind trust and less on strong security practices, I had a good share of doubts that these things can be achieved.

Paraphrasing John Saddington [24], which reinterpreted a famous quote attributed to Leonardo DaVinci, is safe to say that *software is never truly finished*, and this affirmation stands true for the described solution as well.

In the span of a few years, the technological choices that I've opted for will become, most likely, obsolete, and better alternatives will already be on the market, but the design, led by the mandatory principles that are listed at the beginning of the paper, will stand true for a longer period of time.

Thus, the most important thing and the scope of this paper is to present a roadmap that encompasses a privacy-oriented development direction that can guide future implementations.

Sometimes, people do not understand the intrusiveness of the applications they use daily and, in the long term, living without respect for privacy and real ownership of digital data will only benefit the companies that can take advantage of one's preferences, activities, and files to increase their profits by profiting off of the category that should be respected the most: the users.

Despite the fact that a lot of companies grew on the promise of privacy, digital privacy is not a tool that can be slipped into daily activities without a small effort, but it's an effort that brings with it priceless advantages.

The only thing that programmers can do to ensure that the public will react and will switch to more ethical software is to continuously educate the people they come in contact with and to make the choice between big-tech owned apps and open-source apps easier by building more intuitive apps.

*The safer choice is almost never as easy, but, in the long term, it will be, for sure, a lot more rewarding.*

# Bibliography

[1] D. Schmandt-Besserat, "The Evolution of Writing," in *International Encyclopedia of the Social & Behavioral Sciences*, Elsevier, 2015, pp. 761-766.

[2] History, "Printing Press," History, 7 May 2018. [Online]. Available: https://www.history.com/topics/inventions/printing-press. [Accessed 29 May 2022].

[3] Wikipedia, "Digital Revolution," [Online]. Available: https://en.wikipedia.org/wiki/Digital_Revolution. [Accessed 4 June 2022].

[4] T. C. S. Office, "Information Society Statistics - Households 2020," 2020. [Online]. Available: https://www.cso.ie/en/releasesandpublications/ep/p-isshh/informationsocietystatistics-households2020/cloudcomputing/. [Accessed 13 June 2022].

[5] "Salesforce Completes Acquisition of Slack," 21 July 2021. [Online]. Available: https://www.salesforce.com/news/press-releases/2021/07/21/salesforce-slack-deal-close/. [Accessed 15 April 2022].

[6] K. Gullo and M. Stoltz, "EFF to Appeals Court: Apple’s Monopoly Doesn’t Make Users Safer," The Electronic Frontier Foundation, 4 February 2022. [Online]. Available: https://www.eff.org/deeplinks/2022/02/eff-appeals-apples-monopoly-doesnt-make-users-safer. [Accessed 7 February 2022].

[7] F. Lardinois, "Google Drive will hit a billion users this week," TechCrunch, 25 July 2018. [Online]. Available: https://techcrunch.com/2018/07/25/google-drive-will-hit-a-billion-users-this-week/. [Accessed 14 March 2022].

[8] C. Richardson, "What are microservices?," [Online]. Available: https://microservices.io/. [Accessed 13 April 2022].

[9] C. Toma and I. Ivan, Informatics Security Handbook, Bucharest, 2006.

[10] Google, "Demonstrating Quantum Supremacy," 23 October 2019. [Online]. Available: https://www.youtube.com/watch?v=-ZNEzzDcllU. [Accessed 21 March 2022].

[11] Internet Engineering Task Force, "ChaCha20 and Poly1305 for IETF Protocols (RFC 8439)," June 2018. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8439. [Accessed 4 June 2022].

[12] Internet Engineering Task Force, "The Transport Layer Security (TLS) Protocol Version 1.3 (RFC 8446)," August 2018. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8446. [Accessed 6 June 2022].

[13] J. A. Donenfeld, "WireGuard: Next Generation Kernel Network Tunnel," WireGuard, [Online]. Available: https://datatracker.ietf.org/doc/html/rfc9000. [Accessed 8 June 2022].

[14] Internet Engineering Task Force, "QUIC: A UDP-Based Multiplexed and Secure Transport (RFC 9000)," May 2021. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc9000. [Accessed 8 June 2022].

[15] Computerphile, "Chacha Cipher - Computerphile," 19 February 2021. [Online]. Available: https://www.youtube.com/watch?v=UeIpq-C-GSA. [Accessed 4 June 2022].

[16] M. Fichtelmann, "aead-stream Public," [Online]. Available: https://github.com/MaxFichtelmann/aead-stream. [Accessed 14 June 2022].

[17] R. Rivest, A. Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," 1977.

[18] National Institute of Standards and Technology, "Transitioning the Use of Cryptographic Algorithms and Key Lengths (NIST Special Publication 800-131A Revision 2)," March 2019.

[19] "Reactor 3 Reference Guide," [Online]. Available: https://projectreactor.io/docs/core/release/reference/. [Accessed 7 March 2022].

[20] Reactive Streams Organisation, "Reactive Streams," [Online]. Available: https://www.reactive-streams.org/. [Accessed 7 June 2022].

[21] World Wide Web Consortium, "Web Workers," 28 January 2021. [Online]. Available: https://www.w3.org/TR/2021/NOTE-workers-20210128/. [Accessed 7 June 2022].

[22] World Wide Web Consortium, "https://www.w3.org/TR/wasm-web-api-1/," 5 December 2019. [Online]. Available: WebAssembly Web API. [Accessed 9 June 2022].

[23] Wikipedia, "Microsoft and open source," [Online]. Available: https://en.wikipedia.org/wiki/Microsoft_and_open_source#Initial_stance_on_open_source. [Accessed 10 April 2022].

[24] J. Saddington, "Software: Never Finished, Only Abandoned," 1 November 2014. [Online]. Available: https://john.do/software. [Accessed 14 June 2021].

[25] Wikipedia, "ChaCha20-Poly1305," [Online]. Available: https://en.wikipedia.org/wiki/ChaCha20-Poly1305. [Accessed 4 June 2022].

# Appendix

## Table of figures