**Designing a privacy-oriented system for file-storing & file-sharing**

ALEXANDRU – ȘTEFAN BIȘAG

Bucharest University of Economic Studies

ROMANIA

[bisagalexandru16@stud.ase.ro](mailto:bisagalexandru16@stud.ase.ro)

*Abstract:* The privacy of end-users is, most of the times, at the hand of the software development companies, which benefits from the data owned by its customers to increase their profits and extend their operations. The biggest issue with this approach, besides the ethical considerations, is that, in the long run, the lack of privacy will become the de facto standard approach for new applications that are being built. The article tries to formulate a solution to counter the current situation and to create awareness about digital security by outlining a software design for a free, open-source privacy-oriented system dedicated to file-storing and file-sharing.

*Key-Words:* privacy, security, encryption, design, development, files, storing, sharing

**1. Introduction: The state of the market**

In recent years, fueled by the increasing power the big tech companies seem to have on their users, the subject of data privacy became very disputed, both by specialists and enthusiasts.

A few years ago, the European Union adopted the General Data Protection Regulation (GDPR) to help users regain control over the data they are sharing with the companies behind applications, but, even though most companies became compliant and the situation became a little better, at least for the people residing in the EU, users’ data and privacy are still hot topics which are intensely debated.

Among billion dollars scandals, such as Facebook’s affair with Cambridge Analytica, many independent projects were kicked off, marching on the concept of privacy, and, while the idea of offering users a more ethical alternative in regards to their data looks more than appealing, the general user is yet to be convinced, as big tech companies’ popularity on the market still increases.

To make an impact for the general public as soon as possible, trustable, open-source alternative apps should target areas where transferring, storing, and sharing digital data represent the core functionality. The type of systems that interacts the most with data is, by nature, online file-storing and file-sharing systems, commonly referred to as cloud storage: this is where the improvement has to start.

Currently, the biggest proprietary cloud storages destined to 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 only cloud-based storage capabilities, targeting both enterprise and individual users, and are highly integrated with the ecosystems used by their developing company, luring the users in on the promise of usability.

Besides those options, which dominate the market share with an undeniable margin, there is one fairly known outlier, Nextcloud, which promises privacy and security off-the-shelf and targets on-premise deployments to keep the data on a private server, closer to the owner. Even though it features server-side encryption to protect files from unwanted accesses, Nextcloud doesn’t enforce default end-to-end file encryption, leaving users who are not interested in the technology behind their preferred software, to be in the wrong and believe that their files are accessible only to themselves.

However, there’s one particular category of users who, sometimes, become more than that and are, in general, highly susceptible to adhere to the march for a more privacy-focused software: the software developers, the specialists of the domain who have the power to start a pro-policy 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 should, by definition, leverage privacy-related good practices, because the code can be analyzed by anybody. The open-source model gives the developers the chance to become more than users for the software they are using by allowing them to contribute and improve the project by themselves. This way, a project can grow and can reach the general public as a new, better, alternative product which is not exclusively owned by a company and which can be periodically checked by third-party developers to ensure that nobody is taking advantage of anybody.

In this manner, privacy-focused applications can gain more traction, becoming first-choice candidates for a category of users which can popularize them to the general market. Building apps focused on user privacy is a must in the future where big amounts of data will become one of the most valuable assets.

Before blaming the users for not adopting more secure and more ethical alternatives, the users need to have better alternatives.

The change, though, has to 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 have the power to put pressure on the companies that don’t want to change.

2. **Designing a better solution**

A free, open-source alternative for the existing cloud storage services which makes no compromise in terms of security and privacy might be a good start. To improve the situation, in the long run, the education of the public into using more secure and more ethical software should also be taken into account. To ensure that the change will happen, the programmers have to be educated first, so, in the beginning, the proposed system should target them as the dedicated audience.

To sum up all the points enumerated in the previous chapter regarding how an app that promotes and enforces out-of-the-box privacy can have an impact and influence the general public, I composed a list of requirements that defines a minimum viable product.

The most important feature of all is to offer users off-the-shelf end-to-end file encryption and make it as transparent as possible to them that, when used correctly, encryption is the only tool that ensures that files are truly owned by a single party and nobody else can take advantage of them without explicit consent.

The general problem of encryption in terms of adoption is the complexity it introduces through the key management process, which has to be done intuitively both for the main audience of the application, developers, and for the general public that will be targeted afterward.

The approach I’ve chosen was the leverage of colors as identifiers, each password being associated, during creation, with a hex color code. After the password is created and encrypted with the master password, it can be retrieved for encryption or decryption using its associated color. To make it even easier, after a file is encrypted, the entry which identifies that file will be associated as well with the color, to allow users to download files without inefficient password guessing. The whole encryption mechanism is detailed in the second section of the third chapter.

In the software world, privacy is almost always synonym with the open-source movement, which removes almost any doubt about the existence of back doors and known vulnerabilities. The system must be built using an open-source technology stack backed by a numerous community of developers. In the fourth chapter, a selection of frameworks and tools suited for the use-case of the application is presented.

To make sure that the app has chances to be adopted by both organizations and individual users, the architecture should be flexible enough to be considered both cloud-native and on-premises deployable ready by requiring the smallest amount of change possible in terms of configuration.

This characteristic will allow teams to deploy the app in the cloud and scale it according to their needs, while letting individual developers deploy the system at home, on personal mini-computers such as Raspberry PI, without altering the core functionalities. This way, the availability of deployments would match a satisfying level according to its use-case without putting a strain on less potent hardware devices.

This particular topic is explained in detail in the first section of the third chapter.

Additionally, the system should leverage modern technologies and offer a near-native experience on a wide range of devices, to fight against the preconception that security should always come with usability drawbacks.

**3. Application Architecture Overview**

***3.1 System Architecture Overview***

The architecture overview of the system, as it can be observed in the *Figure 1 - Architecture Overview*, encompasses the main functionalities of the system, using the microservice as the smallest independent unit which is specialized in implementing a comprehensive set of features for a well-defined entity that owns its data and interacts with the other entities through HTTP calls which adhere to REST conventions.

The core functionalities are accessible to the end-user through a web application running in the browser, which can be accessed from virtually any modern device.

In terms of communication, the majority of the exchanges happen between the client application and the back-end system through the Gateway microservice which serves the front-end resources and, redirects all the incoming requests to the suitable microservices. Besides internal communication, some of the microservices, Identity and Mailbox, communicate with GitLab Identity Platform and, respectively, with Mailgun to leverage parts of their main functionality that would have either required a complex implementation or too many resources.

The incoming traffic intercepted by the Gateway microservice is encrypted using HTTPS, and it’s rerouted, after decryption, to the internal network using HTTP, to avoid additional certificate configurations. The internal network is secure, being implemented through industry standards, such as Kubernetes and Docker, and does not require encryption.

Besides that, from a security standpoint, the files and the passwords stored by the users are never present on the server in their decrypted format, the whole encryption process is done on the client-side, using either plain JavaScript or WebAssembly, when possible.

***3.2 Authentication***

The authentication process, described in the *Figure 2 - Login Sequence Diagram*, adheres to the OAuth2 standard, leverages the Authorization Code Flow, and relies on GitLab as the trusted identity provider (IdP), allowing developers to access the application seamlessly.

When the authentication process is triggered, a request is sent from the client application, along with a callback URL, to the GitLab IdP servers, which will redirect the user to the GitLab authentication form. After the values are filled, they are sent back to the identity platform, are verified, and, if correct, a signed access token is generated.

The access token is passed back to the front-end application, through a redirect response that relies on the callback URL that was passed on initially. The access token contains basic information about the user, such as the e-mail address, and will be seen as a trusted data provider by the Identity microservice, which will receive the token from the client application.

On the server-side, the Identity microservice will check the authenticity of the token through a request sent to the GitLab identity platform. If the access token is valid, the service will decode it and will identify the user in the associated database. With some of the data retrieved from the database, a JavaScript Web Token (JWT) is created, encoded in Base64, and signed with a secret using the standard HMAC + SHA256 algorithm. After minting, the token is passed back to the client application, ending the authentication process.

The client application will send the JWT back to the server in the Authorization header as a Bearer Token to each subsequent request. The Gateway microservice, which has access to the secret that was previously used, will verify and decode the token, and, depending on which microservice is targeted, will fill additional headers in the original request with the needed data before redirecting it. This way, the secret key doesn’t have to be shared between all the microservices, and the unauthenticated requests will be rejected as soon as they hit the Gateway service and the verification of the token’s signature fails.

When a token is close to expiring, the Gateway service will request a new token from the Identity service and will attach it as a header in the response, along with the data initially requested. The client application will check every response for the presence of this special header and will renew the old JWT each time a new one is issued.

One of the advantages of relying on externalizing part of the authentication system via OAuth2, besides the implicit security benefits of having a dedicated identity provider, is the fact that the registration process is almost identical to the login. The only difference happens on the Identity platform which, when receiving an access token from a user that’s not present in the database, will send an additional request to the Storage service, requesting the creation of the home directory in the file system.

***3.3 Encryption***

The encryption mechanism is one of the most important features of the whole system, being the process around which the whole application was modeled.

Unlike other systems, which encrypt files on the server, using keys not known by the final user in order to protect the file from breaches and unwanted accesses, but not of the system itself, the encryption process here is handled on client-side, using Web Assembly, when possible, to make sure that performance is impacted as little as possible. When Web Assembly is not supported, a slower JavaScript implementation will be used, in an attempt to preserve the main functionality of the system despite the nature of the client device.

The algorithm of choice for file and password encryption is AES-256 which is fast enough and, at the moment of writing, is considered secure and treated as the de-facto standard by governments and by open-source foundations as well.

One other important feature of the system, which has the role to improve the user experience in regards to cryptographic keys management, is using a color-based system to differentiate between passwords and between the encrypted files. Before a user logs in, he must declare a master password and, at least, three other passwords that should include uppercase and lowercase letters, numbers, and special characters. Upon creation, those passwords will be associated with a hex color code chosen by the user from a palette.

The master password will never leave the client machine, so it will not be stored and it’s not recoverable. The user will be informed that he should treat the storage of the master password very responsibly, as it represents the most vulnerable part of the system in case it’s lost. The other password will be encrypted with the master password and stored in the Vault, along with the associated colors.

As described in the *Figure 3 - File Upload Sequence Diagram*, when a user selects a file for upload, he should also select one of the colors associated with a password in the database, along with the master password. The password used for encryption is retrieved from the database using the Vault service, decrypted with the master password, then used to encrypt the file selected for upload, which, in the end, is sent to the Storage service, along with the color of its password and the SHA256 hash computed before encryption.

The download use-case is almost reversed, a fact that can be easily observed in the *Figure 4 - Download File Sequence Diagram*. When a user wants to download a file, the encrypted content, along with its hash and the associated, encrypted password are retrieved from the Storage and Vault services. The user should insert its master password, which is used to decrypt the retrieved password, and then decrypt the file itself. After decryption, the SHA256 hash is computed and checked against the retrieved hash to ensure that the decryption process was finalized successfully, and a download dialog is displayed, to store the unencrypted content of the file on a user-selected location.

***3.4 Deployment***

One of the key features that will be used in the deployment process, as part of the DevOps lifecycle, is containerization. Containerization represents the process of creating a template, called an image, which can be used by dedicated tools to replicate an environment and launch an application with the exact configuration required by the developer. The advantage is that the application will run each time in an identical environment, which is isolated and can be recreated in a matter of seconds.

Each microservice described will be containerized as a Docker Image, in an environment that includes only the necessary tools for the applications to run, keeping the footprint on the system that runs the container as small as possible.

A priority of the system is to be cloud-native, in the first place, but to also be on-premises ready to allow individual users to deploy the application on dedicated servers or personal machines without consuming too many resources.

***3.4.1 Cloud-native approach***

The first approach, the cloud-native architecture, which can be observed in the *Figure 5 - Kubernetes Architecture* guarantees that the system has high availability and possess inherent elasticity by making use of Kubernetes, the most popular and one of the most advanced container orchestration tools.

The Kubernetes configuration files will be managed through Helm, a package manager through which developers can control Kubernetes components with more ease.

The last component of the DevOps cycle will make use of GitLab, GitLab CI, and ArgoCD to achieve the so-called GitOps deployment model, which will ensure that the smallest change inside the source code of any of the components will be integrated into a Docker Image, tested on dedicated worker machines and deployed automatically inside the Kubernetes cluster, mirroring the exact configuration that was requested.

To satisfy the database per service model which is seen as a best practice when developing microservice, the Identity, Vault, and Storage microservices, replicated multiple times inside the cluster as pods will connect to different database instances, ensuring the segregation of data and its exclusive ownership.

The LoadBalancer component will redirect all the incoming external HTTPS traffic to the Gateway service which, leveraging the built-in DNS offered by Kubernetes, will reroute the requests, using the enriching mechanism described above.

Another advantage of client-side encryption and using multiple keys coordinated by the final user is the fact that files can be stored anywhere, without having to live with the fear of brute-forcing, and the PersistentVolume component provided by Kubernetes can be configured to take advantage of the most popular cloud-based storage solutions, such as AWS’ ElasticBlockStore or Google’s PersistentDisk which, even if not open-source, doesn’t need to be fully trusted because of the strong encryption that was already applied to the files.

***3.4.2 On-premises deployments***

The second approach, described in the *Figure 6 - On-Premises Deployment*, targets developers or users who don’t have either the possibility or the need to scale the solution and makes use of the same Docker Images described above, but without the orchestration layer and the DevOps configuration which, even if not required, remain an available option to establish a solid continuous integration and continuous delivery pipeline.

The Docker containers are deployed together using a Docker-Compose configuration, sharing a common virtual network created and handled by Docker, which allows them to communicate directly using the preconfigured container names. For most individual users, having an instance from each microservice is more than enough, as the proposed technologies, described in the next chapter, are optimized for speed in regards to the components’ main tasks.

If scaling is needed, though, an ad-hoc load balancer can be configured using an Nginx container for each microservice type, and multiple instances of the microservices that are under load. The Nginx containers will proxy all the requests and will redirect them to one of the instances, following the load-balancing configuration.

**4. Application Implementation**

***4.1 Domain modeling***

The results of the domain modeling process are simple and easily traceable to the defined microservices, based on their main functionality. The chosen database technology is MySQL, due to its popularity and open-source nature.

The entities, translated into tables as a result of the implementation process, can be observed in the *Figure 7 - Database Tables*.

Data integrity is assured by the relationship between the tables, in the case of the Identity database, which is soft linked to the tables in the other databases through the UUID assigned to the user. This identifier is uniquely defined for each user and is the most important information encoded into the JWT, representing the main user identification mechanism throughout the system. This approach allows the microservices to remain stateless and assume no prior knowledge of the client outside of the current request.

***4.2 Back-End***

The back-end technology of choice is centered around Node.js, a runtime environment based on JavaScript, which is asynchronous by nature, an aspect that guarantees a good overall performance in I/O intensive systems.

To improve the development experience, the other technologies that complete the stack are TypeScript, which is a JavaScript superset that features static type checking and well-defined OOP structures, and Koa.js, the successor of the most popular Node.js framework, Express.js, which comes with an improved, modular architecture that reduces the total size of the production build.

In addition to the usage of these three main actors, which are powerful, actively maintained, and suits the needs of the system, other open-source libraries are used as frequently as possible to delegate general functionality, such as JWT handling, logging, database querying, and use-case specific functionalities, like encryption or push notification sending. There are, of course, many available options, different in terms of interfaces and underlying implementation, but the general rule is to choose libraries that are backed by a numerous and active community that can keep them up to date.

Some of the most popular libraries for achieving tasks that are common for all of the microservices are Passport.js (JWT), Winston & Morgan (logging), Sequelize (ORM).

In terms of third-party integrations, the GitLab Identity Platform is a platform used as part of the authentication mechanism, and its use-case was described in a previous chapter. Besides that, there’s one more third-party integration that can reduce the overall complexity of the system: Mailgun, a mostly open-source project that handles e-mail, and it’s pretty kind with developers in terms of pricing. This integration can be, of course, replaced with a library, such as nodemailer, that can connect to existing e-mail servers through the SMTP protocol.

The core functionalities built using the aforementioned technologies are exposed through REST APIs, which are accessible through the Gateway service, and are natively consumed by the client application, but can be used by other developers as well to integrate secure file storage and file sharing in their custom solutions. The structure of the API, grouped by entities, is described in the *Figure 8 - REST API Design*.

***4.3 Front-End***

The front-end client application is a major component of the system, representing the portal through which a user interacts with all the functionalities implemented by the microservices. In addition to that, the client application implements some interesting features by itself, being a fully-fledged application, not just a user interface.

From those features, the most poignant is the encryption process, which is achieved, in the most favorable conditions, through WebAssembly, ensuring that unencrypted data is never stored on the server.

If WebAssembly is not supported by the host device, JavaScript will be used, with increased attention on how the variables holding unencrypted data are handled.

The implementation of the front-end is based on Vue.js, an independent, open-source framework, which packs a lot of powerful features that allow the codebase to be, at the same, time, concise and maintainable. In addition to that, Vue Material is used to provide beautiful and functional UI components that will seem more than familiar to any user.

The last worth-mentioning touch in terms of front-end development is the packaging of the application as a Progressive Web Application (PWA), an addition that will allow the users to install the client on their devices along with the other native applications, and access it without explicitly opening the browser.

The main screens of the front-end application, the Dashboard, and the File Storage screens, can be observed in the annex section, in *Figure 9 - UI - Dashboard Screen* and *Figure 10 - UI - File Screen*.

**5. Future Development**

The architecture described in the previous chapters is self-sufficient and, if implemented correctly, can produce a usable application that respects the quality checks enumerated in the second chapter.

Paraphrasing John Saddington, 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 currently described solution as well. For every decision that was taken throughout the article, a better decision will exist in the near future and the space for improvement will grow with no doubt.

With this in mind, the most important thing would be, from my perspective, not to amend what was already defined, but to present a roadmap that encompasses a development direction that’s true to the core ideas on which the system was designed, to guide future implementation that might exceed the initial goal.

The most important feature that would help in achieving the main goal of the application, popularizing encryption and making it as seamless as possible, would be the integration of hardware authentication devices, such as YubiKeys, which will eliminate the need for master passwords, reducing the risk of session extraction, if the password is temporarily stored while increasing the ease of use and the general level of security.

In the same direction, targeting security from the authorization point of view, an external policy engine can be integrated to allow the creation of fine-grained access policies that can be already applied for permissions inside a group or for more interactive scenarios that will be developed in the future.

To make sure that the system gains traction among software developers, the initial audience of the application, it’s a must to integrate development-oriented features, such as keyboard navigation, in-browser IDE for code editing, or a built-in admin panel. These will attract developers that can improve the system and contribute to it directly or integrate it more efficiently into their custom applications.

The last development proposal, diversification of used technologies, comes as an obvious choice in the realm of microservices, where each component can be optimized for its main purpose. A concrete example that has the potential to increase the performance of the whole system would be the replacement of MySQL instances with faster, non-relational databases in the systems that are frequently accessed and rely on single table domain models.

Those are, of course, just general or punctual suggestions that can improve the system without affecting its core structure or functionalities but are changes that, once integrated, can make the resulting applications more performant, more robust, and, in the end, more appealing to the general user.

**6. Conclusions**

Designing a system that is both secure and oriented on the user’s privacy is not an easy task and, even though the outcome of the current article represents a well-defined minimum viable product, a complex, failproof system is not a feat that can be accomplished by a single person.

While I was preparing the diagrams, selecting the tech stack, and trying to come up with an intuitive way that can convince the general public to rely more on encryption and less on blind trust, I had a good share of doubts that these things can be achieved, even in the most utopic scenarios, as people are hard-wired to trust and to defend themselves only after the damage has been done. In the tech world, the damage cannot be seen in plain sight and, most of the time, it cannot be seen at all if the companies that are in charge know how to cover up their wrongdoings.

Other times, people do not understand the intrusiveness of the application that 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 profit by profiting off of the category that should be respected the most: the users.

Despite the fact that a lot of companies grow on the promise of privacy, digital privacy is not a tool that can be in daily activities without effort, but it’s a way of life.

The only thing that developers can do to ensure that the mass public will react and will opt for 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 better apps.

# *References:*

|  |  |
| --- | --- |
| [1] | C. Richardson, Microservices Patterns, Shelter Island, New York: Manning Publications, 2018, p. 520. |
| [2] | I. Ivan and C. Toma, Informatics Security Handbook, 1st ed., Bucharest, 2006, p. 421. |
| [3] | J. Richer and A. Sanso, OAuth 2 in Action, Shelter Island, New York: Manning Publications, 2017, p. 360. |
| [4] | M. Lukša, Kubernetes in action, Shelter Island, New York: Manning Publications, 2018, p. 624. |
| [5] | S. Tandel and A. Jamadar, "Impact of Progressive Web Apps on Web App Development," 2018. |
| [6] | P. Borelli, "A complete guide to the Node.js event loop," 6 May 2021. [Online]. Available: https://blog.logrocket.com/a-complete-guide-to-the-node-js-event-loop/. |
| [7] | J. Saddington, "Software: Never Finished, Only Abandoned," 1 November 2014. [Online]. Available: https://john.do/software/. |
| [8] | "GitLab as an OAuth2 provider," GitLab Inc., [Online]. Available: https://docs.gitlab.com/ee/api/oauth2.html. |
| [9] | "GitLab CI/CD," GitLab Inc., [Online]. Available: https://docs.gitlab.com/ee/ci/. |
| [10] | "Helm: Quickstart Guide," The Linux Foundation, [Online]. Available: https://helm.sh/docs/intro/quickstart/. |
| [11] | "ArgoCD: Core Basics," Argo Project, [Online]. Available: https://argo-cd.readthedocs.io/en/stable/core\_concepts/. |
| [12] | A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Grohman, L. Wagner, A. Zakai and B. Jf, "Bringing the web up to speed with WebAssembly," in *The 38th ACM SIGPLAN Conference*, 2017. |
| [13] | M. Massé, REST API Design Rulebook, 1st edition ed., O'Reilly Media, Inc., 2011, p. 116. |
| [14] | B. Yuen, A. Matyushentsev, T. Ekenstam and J. Suen, GitOps and Kubernetes: Continuous Deployment with Argo CD, Jenkins X and Flux, Boston, Massachusetts: Manning Publications, 2021, p. 344. |

*Appendix:*

Diagram

Description automatically generated

Figure 1 - Architecture Overview

Diagram

Description automatically generated

Figure 2 - Login Sequence Diagram

A picture containing graphical user interface

Description automatically generated

Figure 3 - File Upload Sequence Diagram

A picture containing diagram

Description automatically generated

Figure 4 - Download File Sequence Diagram

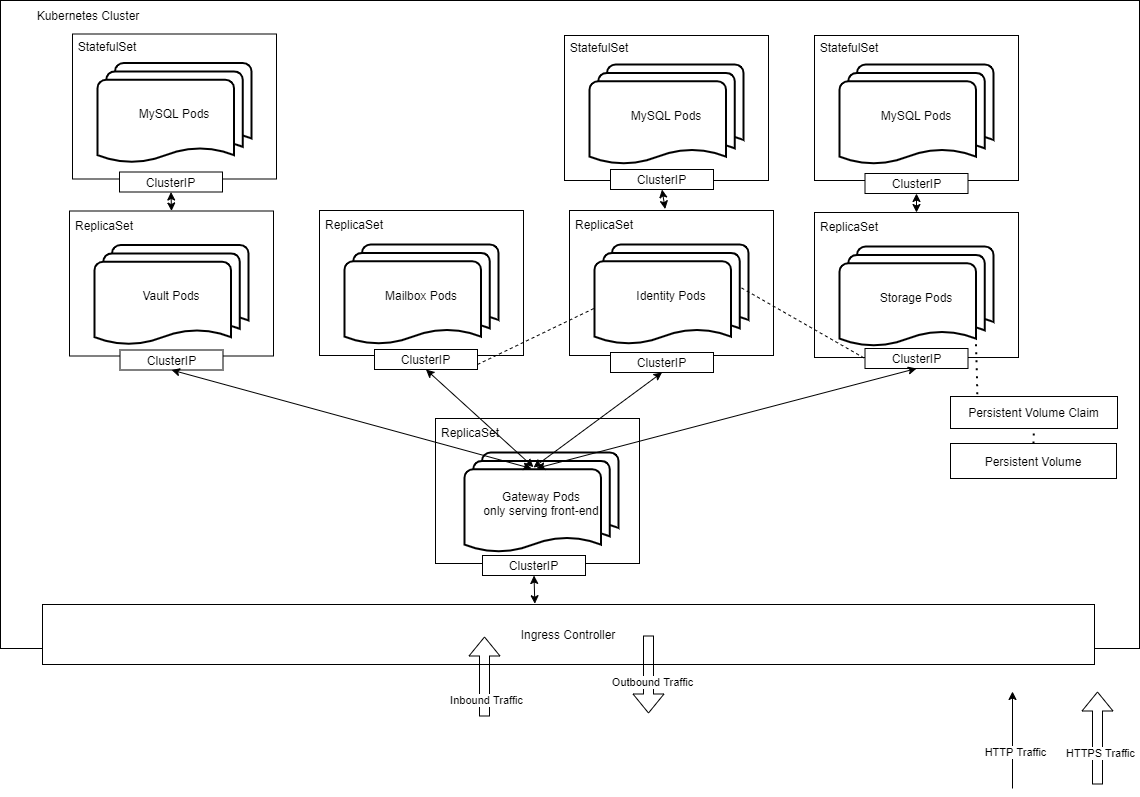


Figure 5 - Kubernetes Architecture

Diagram

Description automatically generated

Figure 6 - On-Premises Deployment

Graphical user interface

Description automatically generated

Figure 7 - Database Tables

Text

Description automatically generated

Figure 8 - REST API Design

Graphical user interface, application, Teams

Description automatically generated

Figure 9 - UI - Dashboard Screen

Graphical user interface, diagram, Teams

Description automatically generated

Figure 10 - UI - File Screen