Use Cases and Logical Architecture

- **XID:** X00189661
- **Name:** Grzegorz Maniak
- **Project Title:** Noise Email

Use cases

Title:	User Registration
Primary Actor:	New User

Story: A new user arrives on the platform and fills out a registration form with their username and password, agreeing to the platform's terms. During this process, the ***user*** generates asymmetric and symmetric keys *(Important: On the Client, the server does not generate these secrets!)* The user's public keys are then sent to the server, which stores them and confirms the registration has been completed, after registration, the user will be redirected to the login page.

Title:	User Login
Primary Actor:	Registered User

Story: A returning user visits the platform's login page and enters their username and password. The platform verifies the credentials and authenticates the user. Once validated, the user is granted access to their encrypted credentials and decrypts them on their machine *(Important: Again, the server never sees any plain text secrets)*, after successful login, the user is redirected to the main dashboard.

Title:	Send Encrypted Email
Primary Actor:	Registered User

Story: A logged-in user writes an email and selects the option to send it **encrypted** *(Note: we will fall back on PGP if the recipient doesn't conform to our protocol, we still need the recipient public key though)*. The email content is encrypted on the client side using the **recipient's** public key. The email is sent to our server, which relays the encrypted email to the recipient's server. Neither our nor the recipient's server can access the email content due to encryption, ensuring full client-to-client privacy throughout transmission.

Title:	Send Unencrypted Email
Primary Actor:	Registered User

Story: A logged-in user writes an email and selects the option to send it **unencrypted**, the email content is **still** encrypted on the client side using the **session** key. The email is sent to our server, which **unencrypts** the email and relays the email to the recipient's server *(In plain text)*. The user stores an encrypted email version for their record keeping.

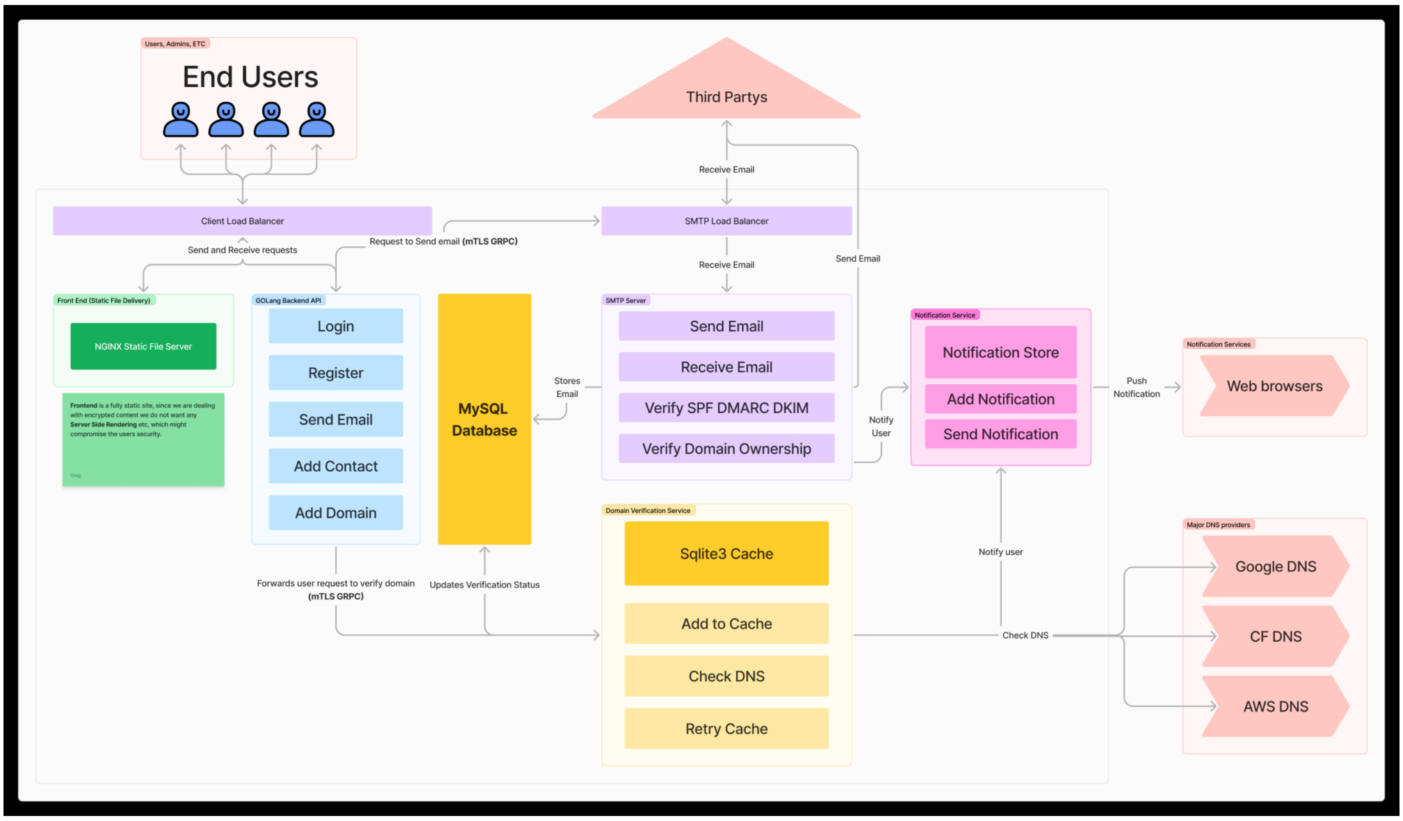
Title: Receiving email
Primary Actor: Registered User, Third Party

Story: A third party requests to send an email to our registered user, they begin this process by fetching our registered user's public key, they proceed to encrypt their own email contents with this key and pass the encrypted email to our server, we verify the email *(DMARC, DKIM, SPF)* and insert it into our registered user's inbox. The user proceeds to fetch this email, unencrypts it using their private asymmetric key (Slow), and re-encrypts it with a private symmetric inbox key (Fast).

Title: Fetch Public Key of Inbox
Primary Actor: External Application or User

Story: An external application or user wants to securely send an email to a registered user on our platform. They start by requesting the public key associated with the registered user’s inbox. Our system verifies and confirms the existence of the registered user and their inbox. The system responds with the user’s public key, allowing the external party to encrypt their message content securely.

Logical Architecture



Note: *Anything marked in red indicates an outside interaction.*

Noise.Email's architecture will adopt a **strategic** microservices approach, carefully dividing functionality across a **few** key, individually scalable components rather than a thousand sprawling array of small services that will explode if you look at it in the wrong way.

I believe that this structure provides the flexibility to scale specific areas of the application as needed, balancing efficiency and maintainability while avoiding the complexity and overhead of managing countless microservices.

Each core service is designed to handle distinct responsibilities, ensuring streamlined, secure operations without clogging the system with excessive inter-service dependencies, which I believe increases security as each specific service is built with the principle of least privilege.

Client Layer

NGINX Static File Server: The frontend is served as static files over NGINX. The setup supports horizontal scaling, since it is just static files. Since we are dealing with encrypted content we do not want any Server Side Rendering etc, which might compromise the user's security.

Client Load Balancer: Both the NGINX frontend and the Go-based backend are hidden behind a unified client load balancer, presenting them as a single cohesive service, this setup allows us to independently scale the front & back end whilst keeping them behind one domain*, noise.email* vs *api.noise.email* and *client.noise.email*.

Core Backend Services

Go-based Backend API: The backend API is built in Go, it is horizontally scalable and provides primary application logic, handling user authentication, email encryption, and database operations. Each instance of the backend has restricted database credentials *(scoped by table permissions)* to ensure access control. It initiates email sends by routing through the SMTP system, managing notifications, and triggering domain verification when needed.

SMTP Server & Load Balancer: The SMTP service is decoupled and independently load-balanced, with an SMTP load balancer that distributes email traffic to available SMTP servers. Outgoing email requests are routed from the backend API over a secure mTLS gRPC channel to the SMTP load balancer, which then directs emails to the appropriate SMTP instance. Incoming emails similarly pass through the SMTP load balancer and, upon processing, trigger a notification to the notification service.

Supporting Services

Notification Service: A single-instance service with a persistent SQLite3 store for maintaining notifications across restarts. It is responsible for storing, sending, and retrying notifications as needed. By operating as a single-instance service, complexity is reduced, and reliability is maintained with data persistence *(It also doesn't have to be fast)*.

Domain Verification Service: Also a single-instance service, the domain verification service manages DNS verification for user-owned domains. It maintains a persistent retry queue to handle re-verification in case of errors. It checks DNS configuration through various providers (e.g., Google DNS, Cloudflare, AWS) to validate user-provided domain configurations. Once a domain is verified, this service updates the main database to mark the domain as verified. This service is the only one with permissions to modify domain verification status, ensuring consistency and avoiding race conditions.

Database and Caching

MySQL Database: The primary data store for Noise.Email, where user data, domain configurations, and email metadata are stored. Each service has dedicated, scoped credentials to perform specific database operations, enforcing least privilege and limiting database interactions.

SQLite3 for Notification Persistence: The notification service utilizes a local SQLite3 store to persist notifications, making it resilient to restarts without requiring external database dependencies.

ETCD for Service Discovery: ETCD manages service discovery, enabling efficient and secure inter-service communication, it also allows us to quickly exchange failing services without having to restart the whole system.

Security and Communication

mTLS-secured gRPC Communication: Services communicate through gRPC with mTLS, securing inter-service communication by ensuring that data transferred between services is both encrypted and authenticated. mTLS will be used for server-to-server verification, where each server verifies connection requests with keys specifically signed and issued by our infrastructure. Since these keys are generated and managed internally, only genuine keys can be used to connect to our internal services.

Scoping of Database Access: Each service is granted scoped access to MySQL tables, allowing only the necessary operations for each service. This limits the potential damage from any misconfiguration or service compromise, enforcing the principle of least privilege. It will also allow us to trace back to which service failed easier.