A close up of a sign

Description automatically generated

**Message Routing Service**

**Technical Design Overview**

# **Introduction**

The Postmark product has primarily focused on the delivery of transactional emails. Now that the product offering is expanded to provide bulk emails, changes are therefore required to differentiate transactional and bulk emails, while facilitating an increase in the volume of messages sent. This Message Routing Service will be responsible for intercepting all email request from the Postmark single and bulk email API’s. These will then be categorized based on defined business logic and sent to the appropriate downstream pipeline.

## **Goals and Objectives**

Key goals and objectives are:

* Provide a highly scalable, available and reliable software architecture.
* Conform to Twelve-Factor App design characteristics.

## **Assumptions**

* The Message Routing Service is not concerned about when and the order emails are delivered when sent to downstream pipelines.
* The single email API will process and submit emails in batches to routing services.
* Since the routing service is a middleware that both Bulk and Single Email API’s interacts, they are required to validate emails before submitting to the routing middleware.  The routing service will only validate that a payload is present and submit for further processing which will disqualify emails that have not met routing business rules.
* Different business rule will be used to disqualified transactional and bulk emails; however, some rules may overlap.
* The email **From** property will be used to differentiate which downstream pipeline emails should be delivered. Each costumer will have a different **From** email for transactional and bulk emails.
* Downstream pipeline will be expose via REST endpoints.

## System Architecture

Message Processor

Message Routing REST API

Fabio Load Balancer

Message Routing Service

Bulk Email API

Single Email API

Consul Service Discovery

RDMS

Mongo Db

**Consul Discover Service & Fabio Load Balancer**

The system uses the Consul service discovery and health checking features, along with Fabio zero-config load balancing functionality to provide a highly scalable and resilient architecture. New instances of the Message Routing Service will register with a Consul master node and Fabio will automatically route traffic to these services. These services can then be torn down and brought up on demand without any additional configuration and complexity. System-level and application-level health checks of each instance of the Message Routing Service will be sent periodically via the Consul agent.

Pros

* No reliance on external tools or processes
* No other services to monitor or maintain
* Highly available by default
* As close to real time as possible
* DNS is easy to use with minimal effort
* Health check is distributed with minimal cluster load

Cons

* The application will contain code to interact with consul, which creates a tight coupling between the application and consul.

**Message Routing Service**

The message routing service is used to categorize, validate and send emails to downstream API’s. It is design for high availability and efficient message processing. The Message Routing Service is separated into logical layers, which provides separation of concerns, complexity isolation and code reusability. The logical layers are separated into Web, Message Processor, Domain and Data Layers. This middleware consists of two main components which are the Message Routing REST API and the Message Processor.

**Message Routing REST API**

The Message Routing Rest API intercepts all email request and forwards them to the Message Processor for processing. It is NON-blocking and uses a fire & forget model to facilitate the high volume of emails. Each request payload received is given a unique identifier that can be used to track status emails sent if necessary. Each revision of API contract will be versioned to ensure graceful service evolution, so clients are not broken once they conform to an active service contract.

Pros

* Non-blocking which improves throughput, latency and responsiveness
* High Interoperability because REST is independent to all application platform or programing languages.

Cons

* Emails are not processed and returned immediately
* Clients have to check on status of messages via a separate endpoint.

**Message Processor**

The Message Processor uses Akka.NET for highly concurrent, distributed and fault tolerant event-driven processing. The Actor Model design pattern is used to delegate concurrent computation processing via actor classes. The actor system is responsible for actor process creation, message routing and parallel processing. Each request payload is sent to a parent actor which creates children actors and orchestrate work to them by using messages. There are four main child actors and each has a single responsibility.

Each message received by an actor is saved to the actor system journal and replayed when the system is restarted. This gives the message processor the ability to continue where it stopped, making it highly fault tolerant, while supporting graceful teardown and startup. Akka.NET also supports pluggable journal stores for relational and non-relational database. The message processor uses a mongoDB journal configuration which provides faster read and write operation than the standard relational database. The Message Processor uses a Front Controller pattern for process flow. The parent actor orchestrates each step by delegating work by sending messages to children. Once the child actor completes processing a message it then sends the result back the parent actor which decides what should be done next. Another design consideration was to use the Routing-Slip pattern to handle process orchestration but given the scope of this assessment that complexity would not add much value. The Actor System also contain multiple recovery strategies to recover from failure scenarios. Failure scenarios such as a database stopped responding or a downstream pipeline suddenly goes down the system can retry again a specified amount of times and at specified intervals automatically. These strategies are defined by parent actors and will require code changes and redeployment.

The Message Processor evaluate two types of business logic. Validation logic which disqualifies emails from being sent downstream and transactional logic. Validation logic are handled by defining validation rules. These validation rule can be enabled/disabled via appsetting.json and will require application restart to take effect. All other validation rule configuration parameters are retrieved from mongoDB so changes are reflected immediately. Transactional logic is handled using the Chain of Responsibility pattern. These are implemented as handlers that can also be enabled/disabled via appsetting.json file and also requires application restart. Internal configuration parameters are also stored in mongoDB database. The order these validation rule and handlers are executed are controlled internally.

Another design consideration was to push the messages to a Message Queue for processing. The problem with this approach is each message would need to be processed linearly, which will have an exponential runtime. This could still be done concurrently, however Akka.NET abstracts the complexity around concurrent processing while also providing built-in fault tolerance.

**MongoDB Database –** This data store will be used as the Akka.NET journal store and to persist any other transactional data generated from the system. It will have replica sets for redundancy and high availability.

**RDMS** – will contain relational data required by business domain.