Certified PDF Api

Release 0.1

Gysen

Jul 05, 2022

INTRO

V1 of our **Certified PDF Api** provides the certified distribution of invoices from supplier to client.
It installs an intermediary verification layer that fulfils a double purpose:

- **Verified source**:

  Clients can declare a whitelist of suppliers through a smart contract, from whom they receive periodical invoices as a measure of protection against phishing.
  The smart contract filters invoices based on their supplier IBAN, and therefore decides whether or not an invoice transfer is allowed.

- **Immutable invoices**:

  All invoices are stored as encrypted PDF files in MongoDb, after which we generate and store hashes for each document on a Hedera topic to prevent tampering.
  Once the invoice is stored, it is immutable and cannot be changed without being noticed.

This documentation site describes our process flows and technical api architecture.
We rely heavily on multiple rounds of security analysis to determine and eliminate potential security vulnerabilities.

Note: This project is under active development.

CHAPTER

ONE

CONTENTS

1.1 Source verification

There are different phishing techniques that allow insincere attackers to trick clients into paying wrongful invoices; through fake emails, redirects to malicious website url's and misleading domains. If not for the reason of security, the practise is confusing and all-round annoying for the receiving client. Spam filters allow black lists against insincere emails, but managing blacklists is not a clean solution to the problem.

For each receiving client, our api creates a default client smart contract upon user registration, which maintains a whitelist filter of supplier IBAN numbers. The set is empty per default, which implies that clients are themselves responsible for composing their own whitelist. They should be the only ones to decide who is allowed to send them an invoice, without requiring to declare who is not (blacklist). The same contract whitelist is then used by our server to verify whether an invoice pushed by a supplier is allowed to be transferred to, or requested by the client.

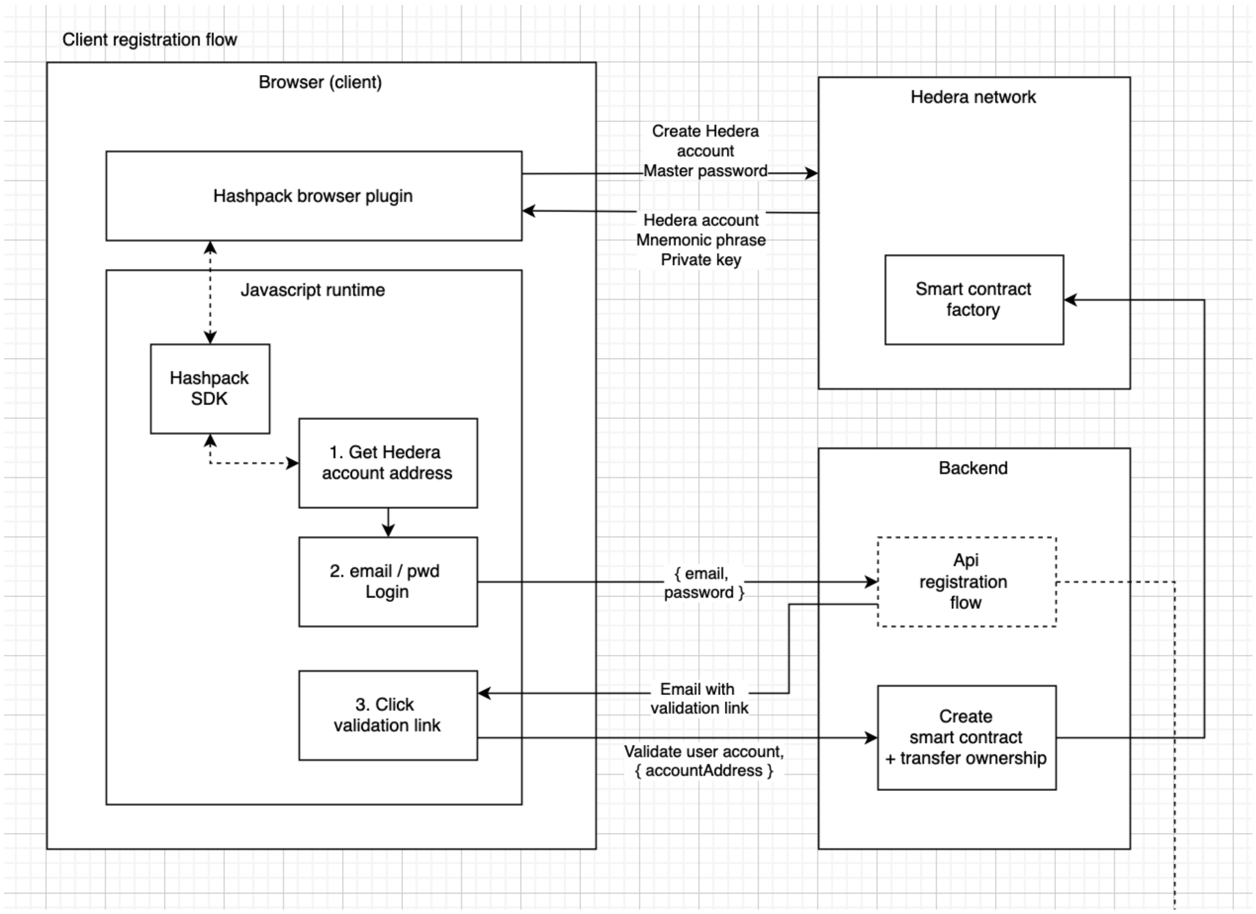
1.2 Immutable invoices

All supplier invoices entering the system are stored as encrypted binary files in MongoDb, along with metadata about the invoice that is needed for further processing.

We protect our database against tampering by hashing database documents with a secret and storing them on a Hedera topic. Whenever we request data from the database, we can verify whether the document has been tampered with by verifying the hash.

Warning: We have yet to consider how to handle the case of tampering gracefully, in case it occurs. Since invoices can never be updated, rolling back to a previous version would be a viable option.

1.3 Client registration flow



1.3.1 Hedera account

The first thing users need to do is to create a Hedera account through the Hashpack browser plugin. It is much better to create the Hedera account from the client rather than from our server, because the user's private credentials (master password, private key, mnemonic phrase) are never exposed outside of the user's local machine. The procedure for registering a Hedera account can be found here: https://www.hashpack.app/post/how-to-create-your-first-account-with-hashpack.

All interactions between client and the Hedera network should be unique handled from the client without passing through an intermediary central server.

1.3.2 Register website account

Only after the user has created a Hedera account, (s)he will be able to register an account to our website.

> **Warning:** Consider allowing Google / LinkedIn etc, given that the user allows "email" permission

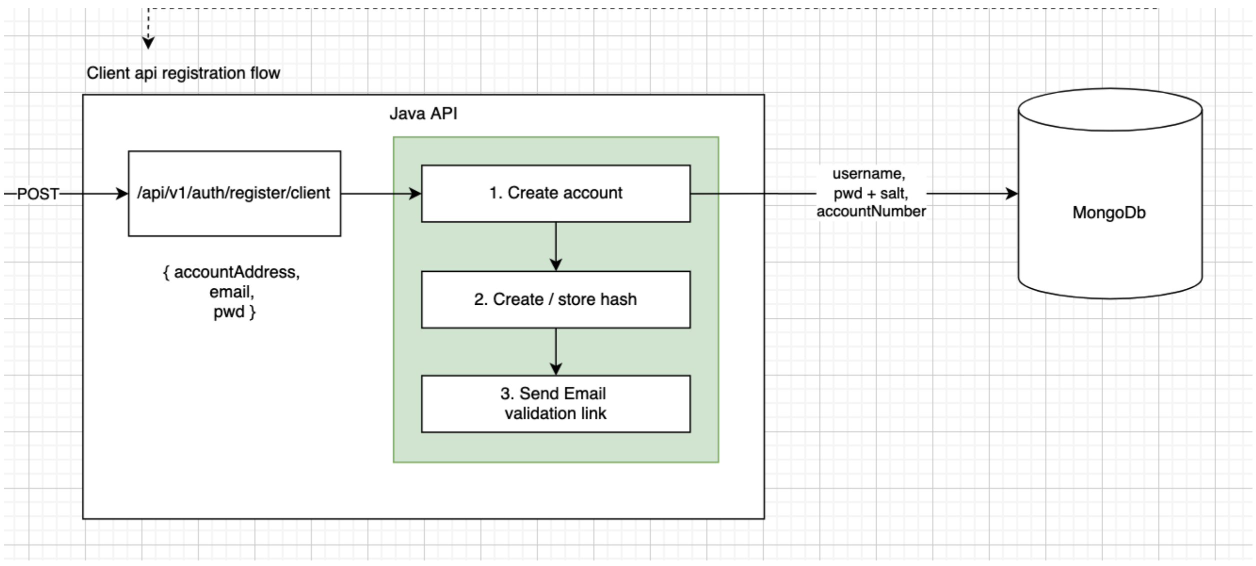
The preliminary creation of a Hedera account is necessary first, because our backend needs the user's account address for transferring ownership of the user's smart contract after the email address was validated. In the backend we store the user password encrypted with a slow hashing algorithm (bcrypt), and an additional salt that is included in the hash to prevent easy recognition of duplicate passwords in the database.

The password that we store is merely useful for logging in into our website to visit user specific content and downloading invoices. In no way does it expose the Hashpack master password that is used for configuring the whitelist configuration of suppliers, since this is done through a direction connection from the client to the Hedera smart contract.

The login flow to our website follows the general oAuth2.0 flow. Even in the very unlikely event that the user's access token gets stolen during a brief time window, the smart contract's ownership is safe.

1.3.3 Backend registration flow

The registration flow in the backend follows a standard flow:

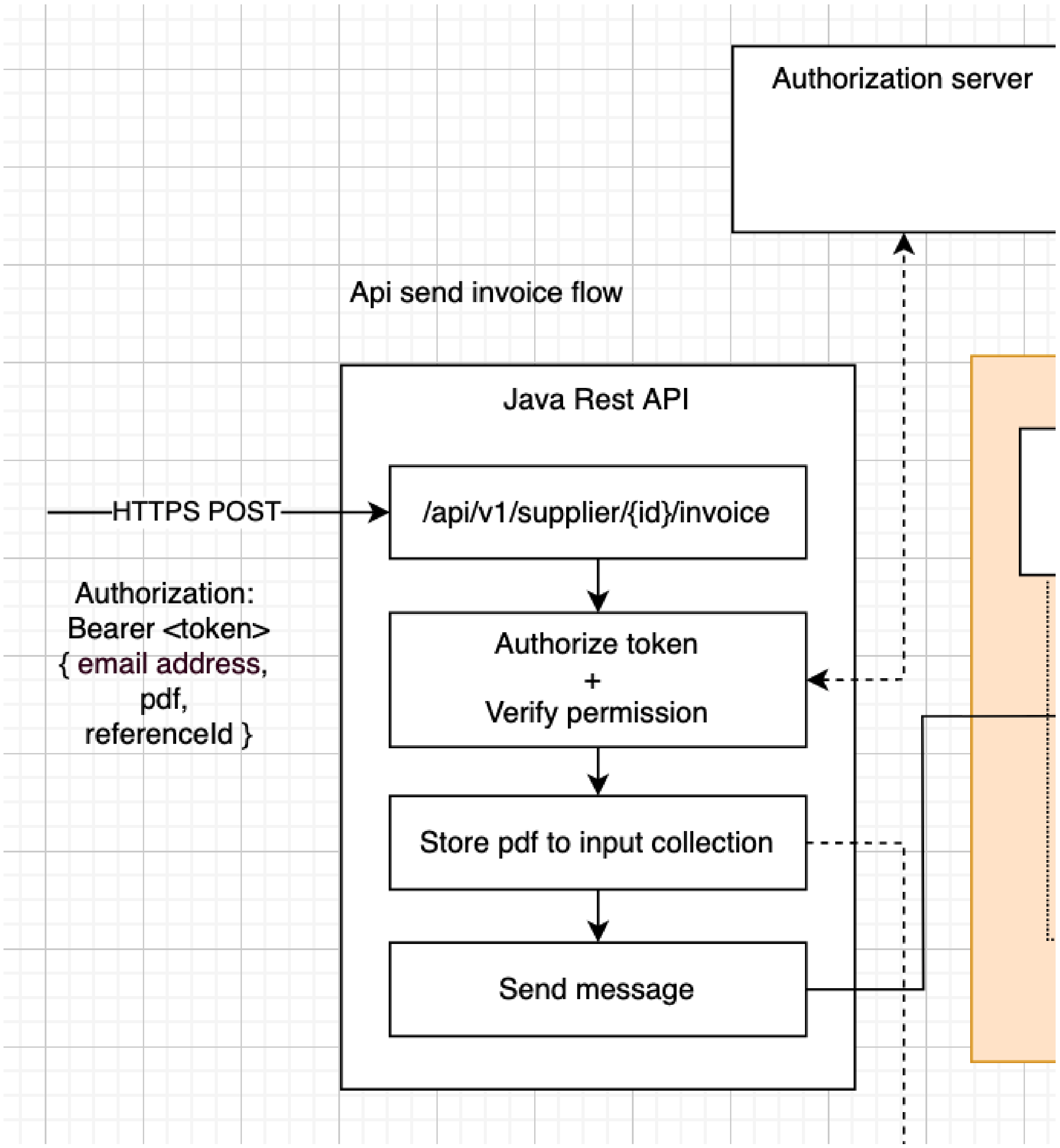

Client api registration flow
Java API
POST
/api/v1/auth/register/client
{ accountAddress, email, pwd }
1. Create account
2. Create / store hash
3. Send Email validation link
username, pwd + salt, accountNumber
MongoDb

1.4 Supplier registration flow

Suppliers should prove that the IBAN they provide during registration is theirs. The client will use the IBAN number to whitelist the suppliers they receive PDF's from.

> **Warning:** To be defined: currently unclear

1.5 Invoice api



1.5.1 REST API

Suppliers should send their invoices as https POST multipart requests. The request should contain the PDF they want to publish, along with a POST body containing meta data. I.e. the client email address they want to send the invoice to, and possibly a company-specific internal invoice referenceId. The supplier should send a bearer token in the request's Authorization header; this is the access token that they obtained during login, and that was provided by our Authorization server following the oAuth2 token flow.

1.5.2 Token format

Although opaque tokens would have preference (they don't expose information embedded in the token), it requires a round trip to the Authorization server for every incoming request. It is more performant to use JWT's, which can be parsed locally at the resource server (see oAuth2 architecture) without requiring a remote HTTP call. It could in theory be possible to encrypt the token to hide the data it contains, but there isn't really need for this since the JWT won't contain any secret identity information. The only thing that matters is that the JWT hasn't been forged /

1.5.3 Client SDK

Following the oAuth2 flow, access tokens are generally limited in time. Since we want our flow to comply to the maximum security guidelines, we respect brief expiration times. This also means that the clients need to implement the oAuth2 refresh token flow to request new access token regularly. It is generally known that this is quite the hassle, therefore we provide a client SDK that eases the interaction with our API.

Warning: The design of the client sdk needs to be analyzed.

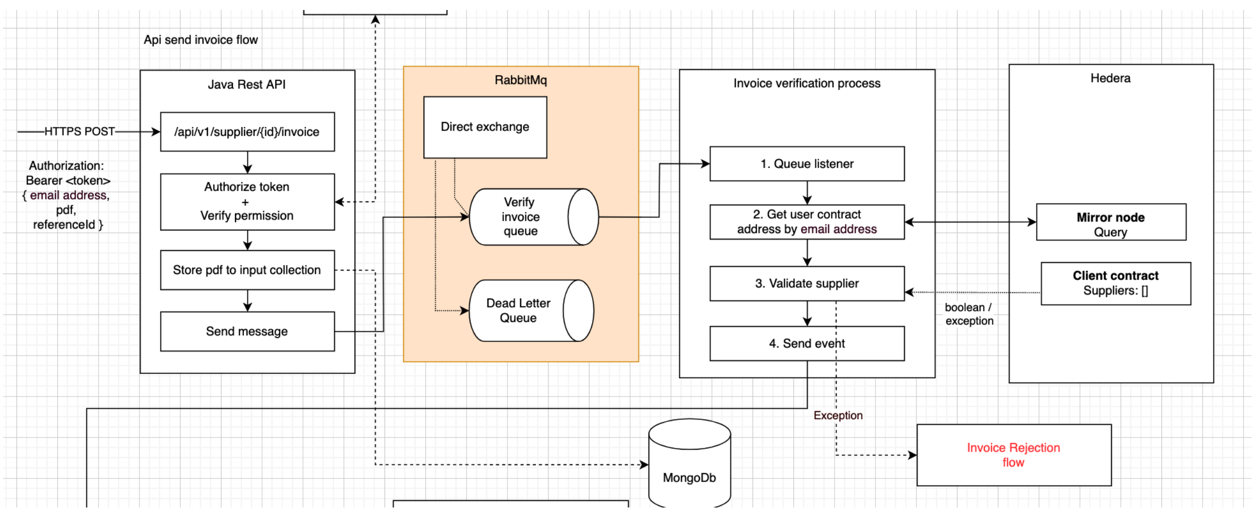
1.5.4 Store input invoices

All pdf's that enter the system are immediately stored in a collection as encrypted binaries. We use a two-way encryption, since the pdf's need to be decrypted when the user wants to view or download them. One reason is that it is better not to send full PDF documents as binaries through Rabbit queues because of their message size. It is better to store all documents in persistent storage, while sending meta data about the stored document through the queue. This requires less bytes to be transferred and instead allows for lightweight messaging.

Another reason is that not all PDF's will be validated positively. We keep track of all PDF's that entered the system in order to handle rejected invoices gracefully based on the reason of rejection.

Warning: To be decided if we need to generate proofs for all input documents (I think the answer is yes).

1.5.5 RabbitMq



To manage high loads of POST requests, we are required to use a queueing system like RabbitMQ (AMQP protocol) to handle back pressure (data buffering between processes). Although the REST api that we expose uses multithreading, it is insufficient on its own under high pressure. The number of rabbit listeners can be configured in Java to allow parallel processing.

Note: Although our system would benefit from using a reactive API, Java SDK19 comes with the loom Project that supports Virtual Threads in Preview mode. At least initially it is fine to write code in blocking fashion, and use Virtual Threads once they're ready to use in Production.

1.5.6 Bulk uploads

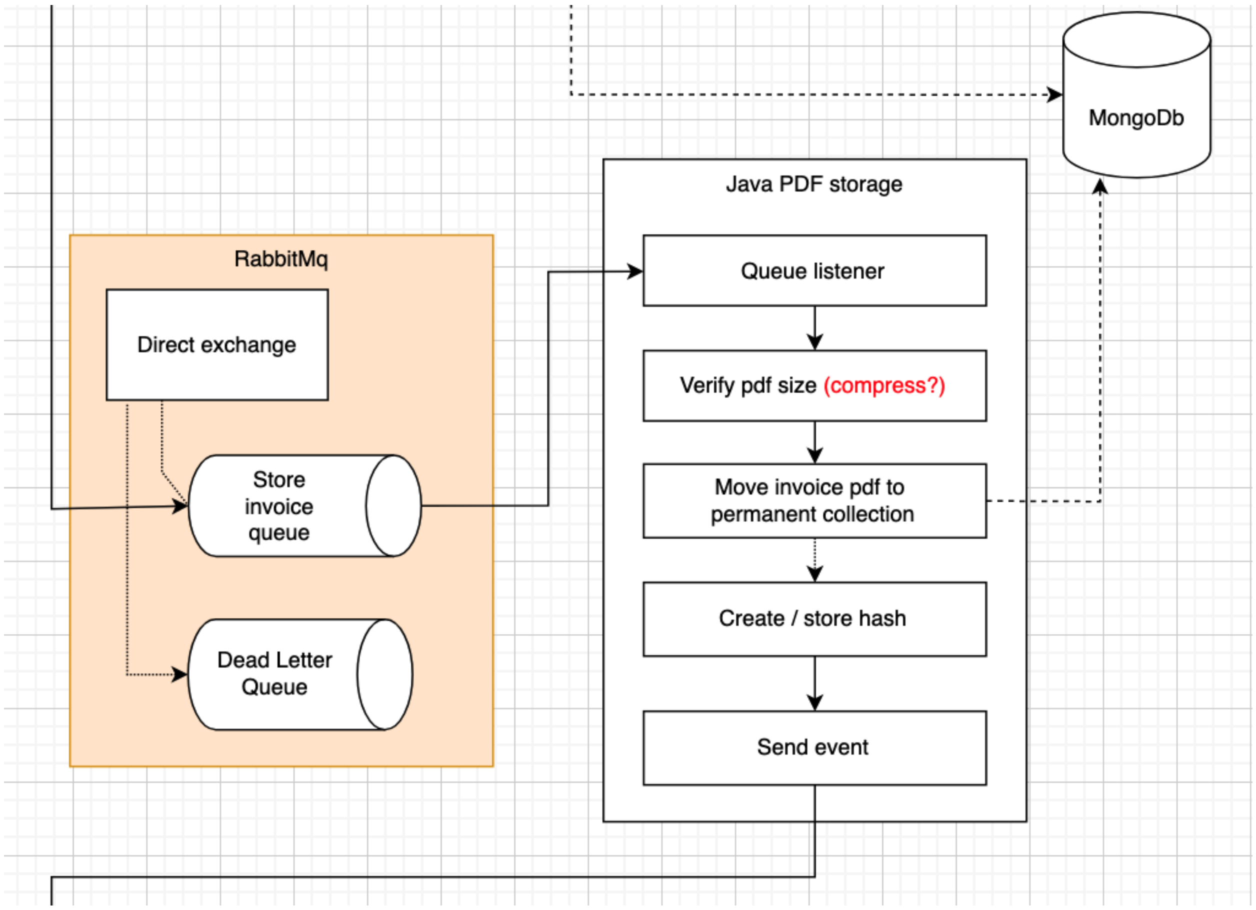
We define 3 request types that can be sent to our api, categorised by the heaviness of the load (low, average, high). The lowest request type involves a manual / low number of expected requests, whereas the highest involves bulk / mass uploads.

Note: Initially we will focus on the manual upload implementation first, as contains the easiest architecture. We will gather metrics, and can decide later whether bulk uploads are worth it.

We enforce limitations on the load by implementing a rate limiter.

Warning: Further measures will be taken to ensure that the number of requests can be controlled. If we were to upscale in order to perform mass uploads, it is important to ensure that uploads from one supplier don't impact the performance of another. To be determined later.

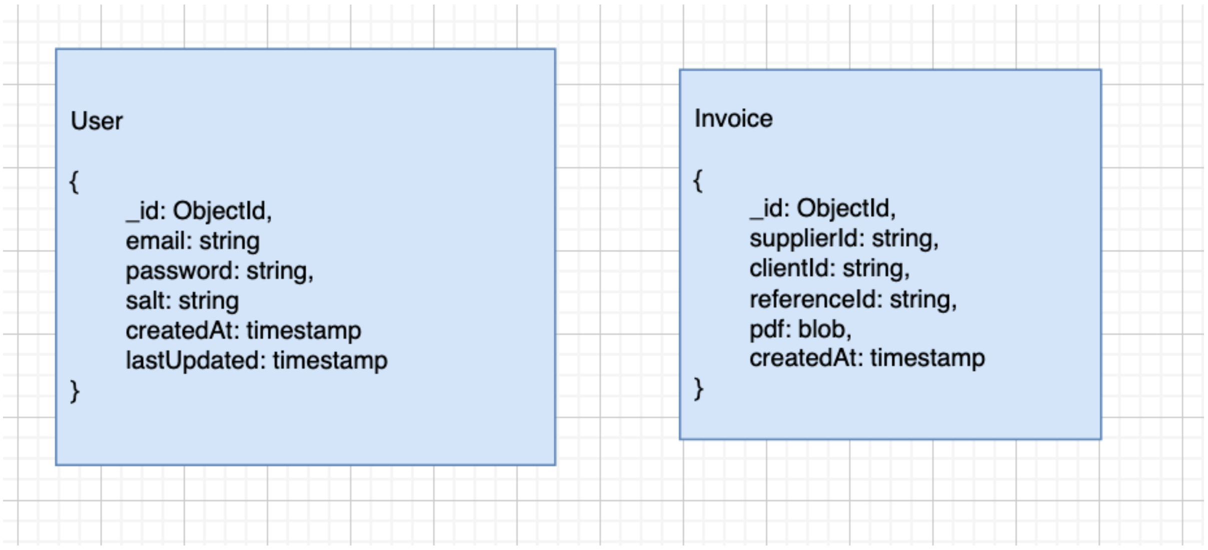
1.6 Invoice storage



Before storing the PDF, we verify if the pdf size is acceptable (eg. 100Kb), and may apply compression if necessary.

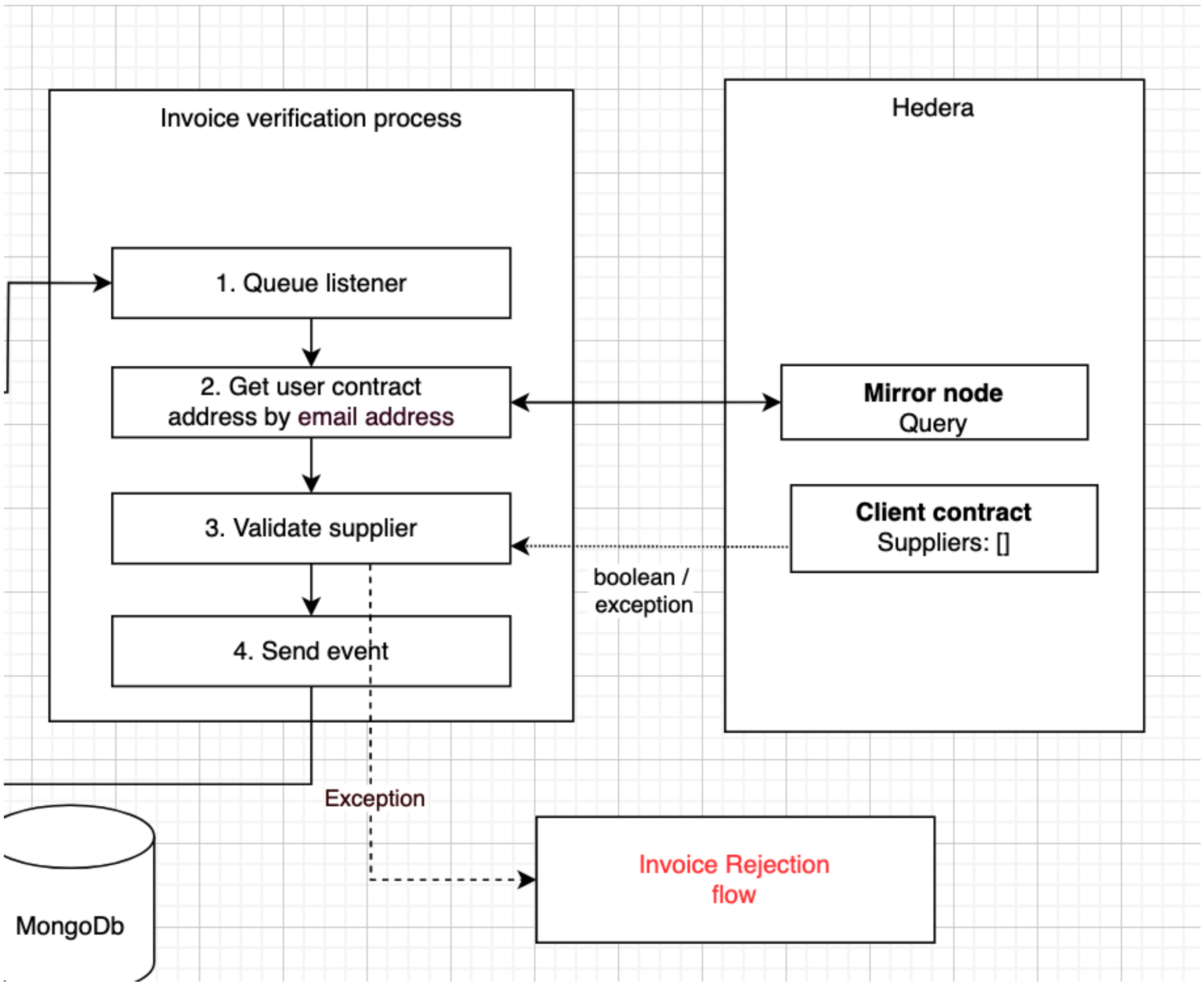
Warning: Compression degrades the PDF quality. The easy solution is just to reject PDF's that are too large which comes with its own downsides for suppliers. To be decided.

Theoretically it is possible to store invoices as embedded documents within User documents. In our case we store them in a separate collection because we will generate and store proofs on Hedera for each invoice separately. When the user wants to view / download an invoice, the hash will be verified.



Warning: To be decided whether or not we will use GridFs to store the documents. MongoDb allows storing files directly in a document up to 16MB, which is plenty for our use case. The trade off between the two has yet to be determined.

1.7 Invoice verification



1.7.1 Find client user smart contract

Based on the the email address of the client user (the receiver of the invoice), we query the Hedera network using an HTTP call to their mirror nodes in order to find their client smart contract address: https://testnet.mirrornode.hedera.com/api/v1/ Hedera mirror nodes are read-only nodes; they get populated with data from the main nodes through a gossiping protocol, in order to improve the performance quality of the Hedera network.

Warning: We should consider the time it takes for mirror nodes to be populated from the main nodes. If an invoice is sent while the user smart contract doesn't exist yet, then it means the verification can't be done. This can be solved by storing these invoices in a dedicated collection to be processed by a scheduler at a later time.

1.7.2 Validate supplier

We verify if the supplier has been whitelisted by the client user by making a call to their smart contract. Reading from the smart contract doesn't require a user's private key, nor does it cost transaction fee. If the supplier was not found, then the message should be sent to an exception queue, to gracefully handle requests that were declined.

Warning: We should consider what to do with clients that don't have a smart contract. Did something go wrong during contract creation? Should we send an email to receivers that haven't registered with our application as promotion?

Warning: We should consider what to do with invoices that get rejected. Should we inform the supplier of rejected invoices? As discussed, if a user doesn't have an account yet, we will still send them an email with different template.

1.8 Microservice architecture

We apply a microservice architecture, which means that we run every service as an isolated process.

The first reason is that failure of one process should not impact another. If, for example, the process of PDF storage fails for some reason and requires a restart, this shouldn't impact user registration, or the registration / verification of incoming PDF's received from suppliers.

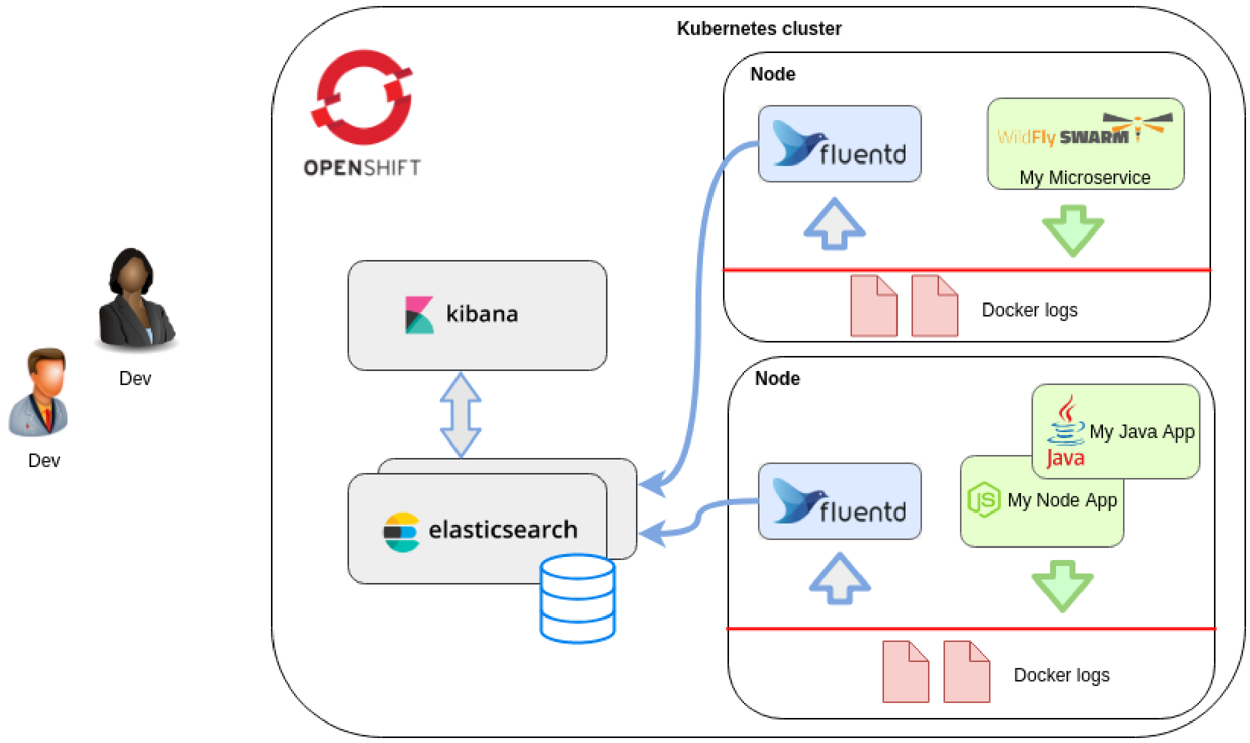
The second reason is that microservices allow us to dynamically distribute and scale the processes independently. One process may be more cpu intensive than another, and one process may require more time to process compared to another. Depending on the cluster technology used, we will be able to choose for an optimal setup in which we can monitor and scale different processes dynamically depending on the need that we observe.

In order to save on initial cost, it should be sufficient to use a single database.

1.9 Logging

For logging, the EFK (or ELK) stack is a common tool for professional log analysis. Since microservices are run in a distributed environment, we use a centralised logging system.

EFK (ElasticSearch - FluentD - Kibana) is used to store log messages in an ElasticSearch database (based on Apache Lucene, performing full text query searches), while Kibana can be used to visualising logs.



FluentD is a log collector that listens to aggregated application log streams (from different applications), and can be configured (among others) as a whitelist for filtering logs based on string patterns.

1.10 Usage

1.10.1 Installation

To use Lumache, first install it using pip:

```
(.venv) $ pip install lumache
```

1.10.2 Creating recipes

To retrieve a list of random ingredients, you can use the `lumache.get_random_ingredients()` function:

The `kind` parameter should be either `"meat"`, `"fish"`, or `"veggies"`. Otherwise, `lumache.get_random_ingredients()` will raise an exception.

For example:

```
>>> import lumache
>>> lumache.get_random_ingredients()
['shells', 'gorgonzola', 'parsley']
```

1.11 API