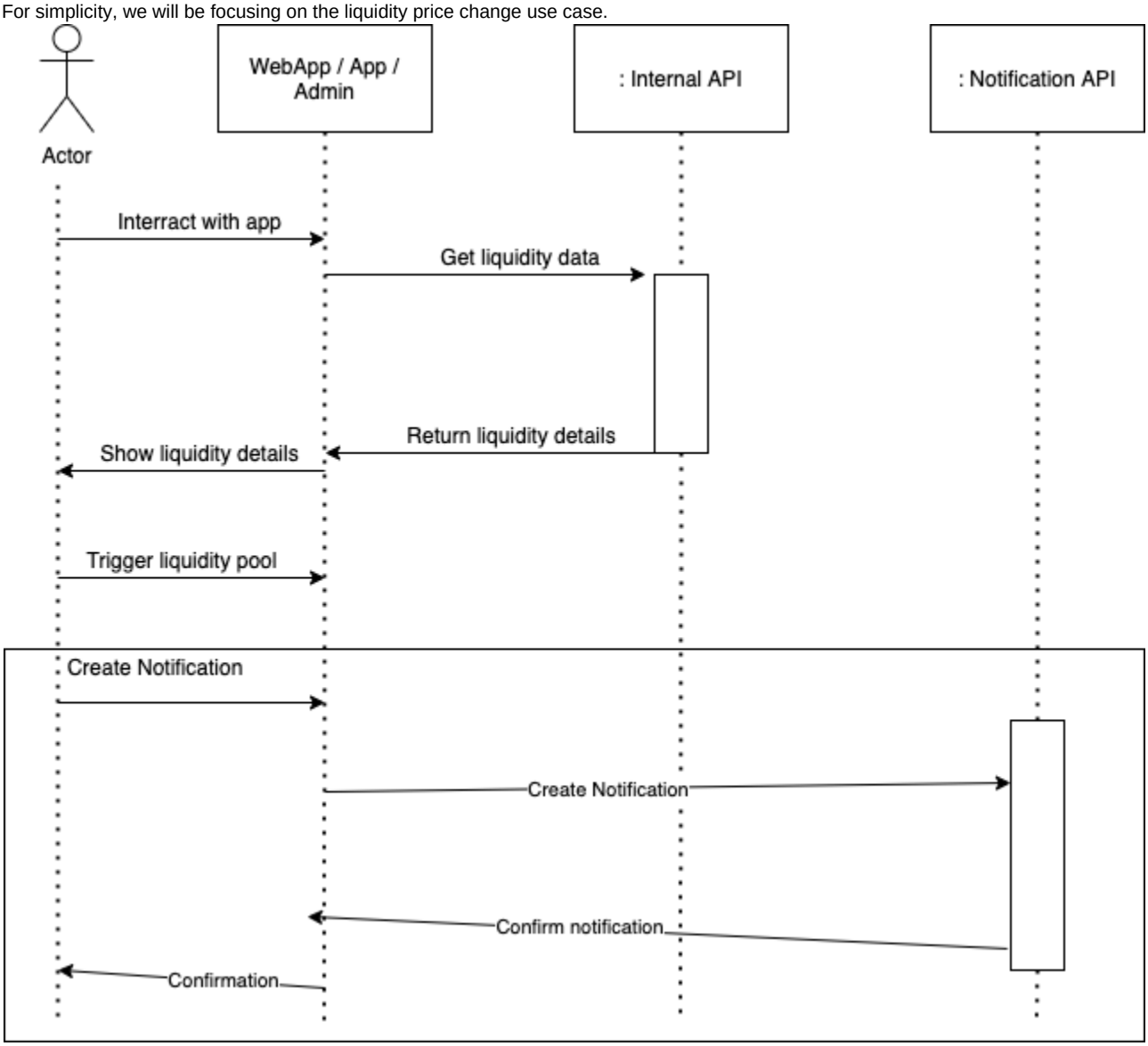
Architecture flow notification service

This doc can serve as a reference document that highlights a Notification Service infrastructure.

I will be using “Notification Service” as an umbrella term to cover the different components that we will need to run a fully functioning notification service on production.

Notification services are widely used nowadays in any product. They are useful whether you’d like to be notified of a change in price (ideally a price drop) where you are interested in, or whether you’d like to be notified if there’s a new job specification available for a job search criteria that you have specified.

For simplicity, we will be focusing on the liquidity price change use case.



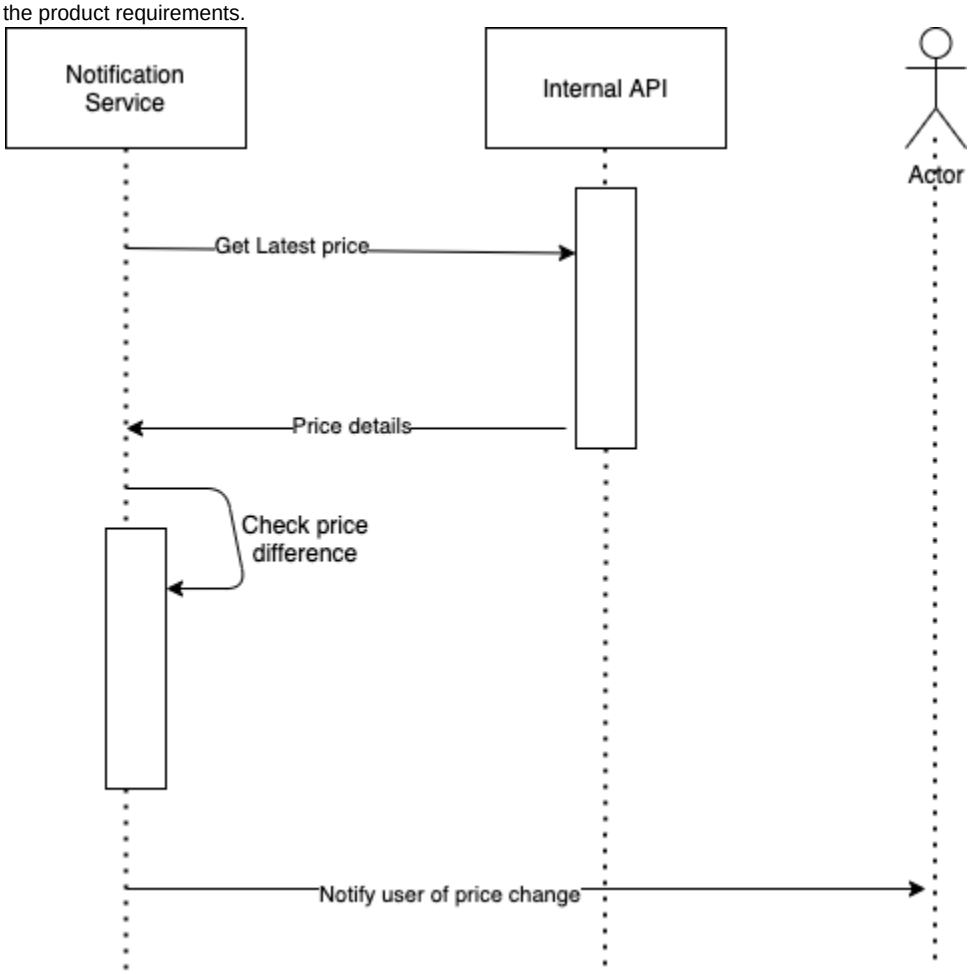
After a actor views a product, the next option is to either buy the coin or to “watch” the liquidity pool in order to get updates if the price changes in the future.

Use cases

The architecture we will come up with in this doc will be able to cover the price change and the availability use cases. However, it is designed in such a way that it can be modified to cover other use cases of your choice.

Price change

When you are looking for a coin/token onto different liquidity websites, pools, etc and you are not sure yet whether you are buying the coin/token in its current price, you might like to be notified if the price has dropped so you can buy within the price range that you desire. We are going to scope down our example in this post to cover this use case. The system will be designed such that this use case can be changed according to the product requirements.



Coin / Token price to high

If the coin/token that you are looking for price is too high OR for example liquidity risk is to big, you'd like to be notified when it is suitable for you based on levels you will select. Depending on the implementation, the same Price change logic can be used for this use case. For example, if the Latest Price is Null, then it means that something goes wrong on to the microservice.

Product requirements

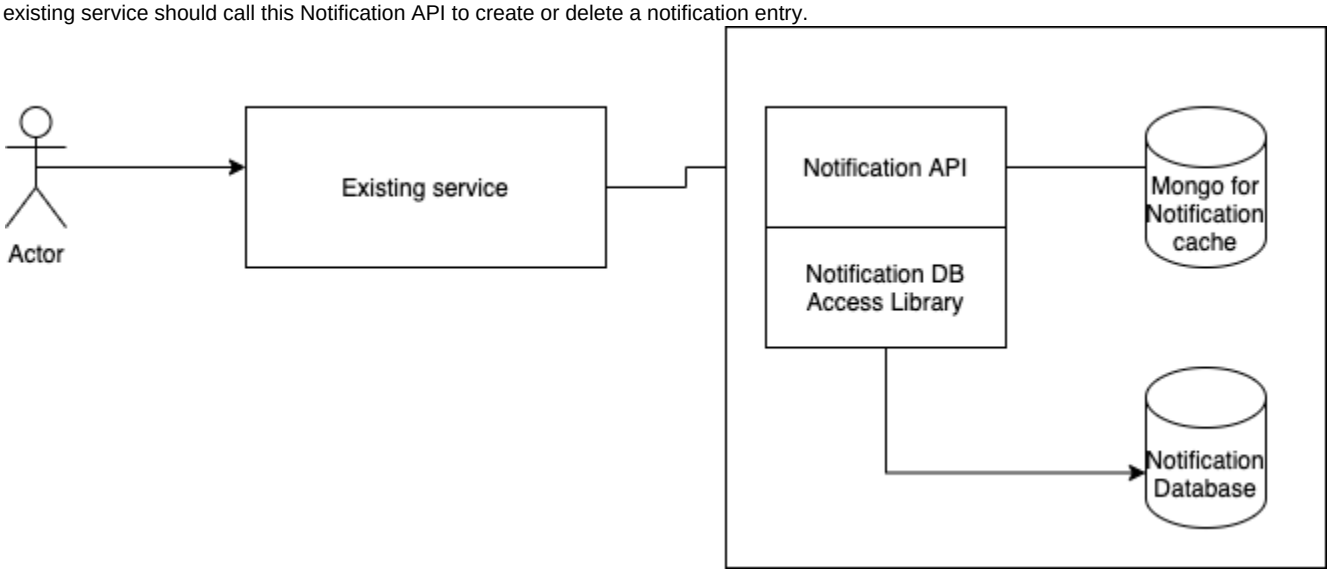
- Logged-in users should be able to create a notification either to "watch" or follow a liquidity pool, coin/token price to get notified if there's a price change or risk of pool. The "watch" functionality is available in the account area or via email or via message hook.
- The service should be able to send an email or a push notification to users if there is a price change or other event to the watched entity
- The user should receive a list of all notifications in the email. For push, there is only one notification at a time based on a set of rules. For example, if a user is watching multiple liquidity pools, we can choose to send a push notification only about the liquidity pool with the biggest risk.

Design goals

- The notification service can be integrated into any existing services
- The notification service should be isolated into its own set of components from its main service.

Main component: Notification API

This will be a vital component of the entire Notification Service. We should be able to create and delete a notification entry via an API. The existing service should call this Notification API to create or delete a notification entry.



Note the “Notification DB Access Library” which will be handy later when we will need another service to connect to the Notification Database.

Notification criteria can be the liquidity risk or the selected coin/token. Each criteria entry is saved in the database. The created entry will help the system understand when we should notify the user.

The API schema could roughly look like the following:

- *createNotification(APIKey, ProductID, userID)*
- *deleteNotification(APIKey, NotificationID)*
- *getNotifications(APIKey, userID)*

The database schema will require the following:

- NotificationID. Unique notification ID. Can be a UUID or an Integer.
- UserID. Logged-in user who created the alert.

The schema should include notification parameters like the following (depending on the product or service. In this example we will consider a liquidity pool.):

- LiquidityPoolId: Liqiuidity pool that we are interested in.
- Latest Risk. Most recent risk scoring found for the searched pool.
- PreviousRisk. Optional field to keep track of the previous recorded risk. Useful for debugging and investigating potential risk issues raised.
- LastUpdated. Timestamp as to when the entry last updated.

Implementation

This can be a RESTFul or a gRPC service with NodeJS, Python, Java, Go, or C# with a NoSQL datastore and a caching system. We expect mostly create and delete for every notification row. Reads are expected if our product allows users to see the list of notifications created.

We will cover the different trade-offs when choosing a Database for our Notification API in the later sections.

Optional: User data component

Ideally, the notification API should only keep reference of the UserID. Users’ PII (Personally Identifiable Information) data like their names, emails, and phone numbers should be from another API for clear separation of responsibilities.

Batch jobs

A straightforward approach if your service relies on third-parties for pricing is to have a batch job that runs periodically to check if there are changes to the price of the product a user is “watching.” In order to check the cheapest price, the batch job needs to call an internal API that returns the price of the product.

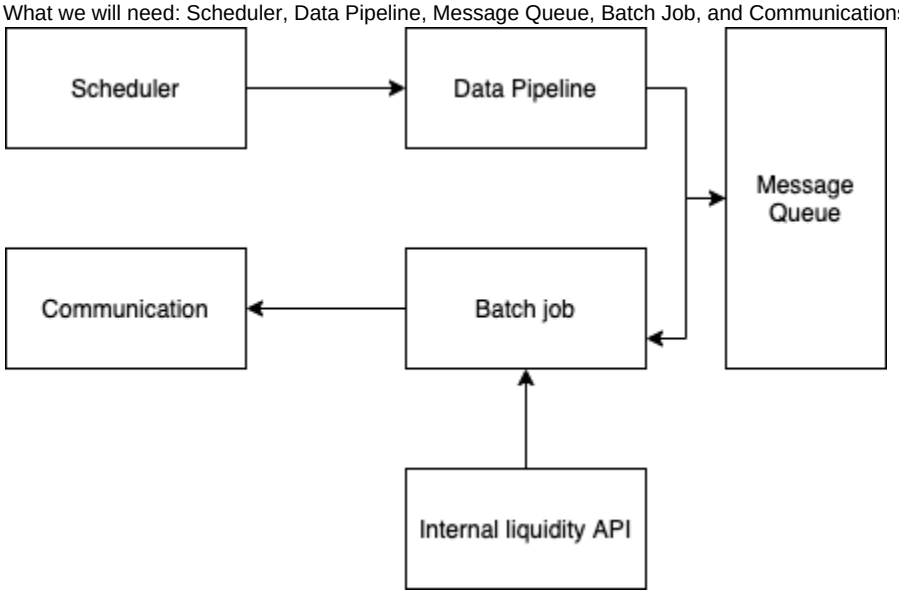
Personal Pros of Eugene:

- Good coverage. It will cover all the search criteria stored in the database.

Personal Cons of Eugene:

- More notification entries mean more API calls to your internal API. Consider optimising the internal API that will be called the batch job.
- Requires a partial or full database scan. Regardless, it’s best to run the scan during off-peak hours.
- Requires more running components.

What we will need: Scheduler, Data Pipeline, Message Queue, Batch Job, and Communications service.



Scheduler

This is a component that we can use to run our data pipeline, batch processing, and other services periodically following a specific schedule, e.g. daily or weekly.

The start time of the data pipeline and batch processing depends on the following:

- Off-peak time. We’d like to do the Notification service database scan when there is less traffic.
- User’s time zone. Time the scheduler accordingly to when the user is awake to receive the notification. This will most likely equate to a better chance of the user seeing the notification, especially for push.

Scheduling strategies to consider:

- All notifications are covered per day.
- Alternate schedule between regions or availability zones (AWS) per day. For example, if your services are deployed in four AWS regions you can alternate the regions covered per day with *ap-southeast-1* and *ap-northeast-1* on one day, then *eu-west-1* and *us-east-1* on another day.

Implementation tip: Good old Cron Job or an *AWS Cloudwatch Event*.

Data pipeline

The scheduler tells the data pipeline when to start. It scans the database, groups the notifications according to a set of criteria, and publishes these notifications to a queue.

This component reads from the Notification API Database.

These are some strategies to consider when reading from the Notification API Database:

- Full Database scan. The batch job will read all the rows in the Notification API database. This is easier to implement for cloud-based storage like *AWS DynamoDB* that allows you to adjust the read capacity when needed, though it can incur costs. RDBMS will do the job, too, but will be better off with a read-only slave that is meant solely for the batch job.
- Partial database scan grouped by regions. The batch job reads the entries based on the region it belongs to. For example, if the scheduled job is in *ap-southeast-1*, then all notifications from users in Singapore, Malaysia, the Philippines, and others within the ASEAN region will be processed.

After reading the notification entries in the database, the data pipeline will then publish these entries to a message queue.

Implementation tip: Use AWS Data Pipeline, Spark, or similar services that can read from a database and group large amounts of data.

Batch jobs

Batch jobs are subscribed to the notification entries topic in the message queue. These jobs will pick up the messages from the notification queue.

The batch job will call the Internal Pricing or Availability API to get the latest prices based on the notification criteria gathered from the database.

- Compare the recent price from the API with what was recorded in the database.
- Based on the price difference or whether the product is available or not, decide whether to notify the user or not. Notifying the user means sending a request to the Communications component.
- After we have successfully completed the request to the Communication component, we will update the notification entry in the database. The LastUpdate and Prices fields will need to be updated at this point.

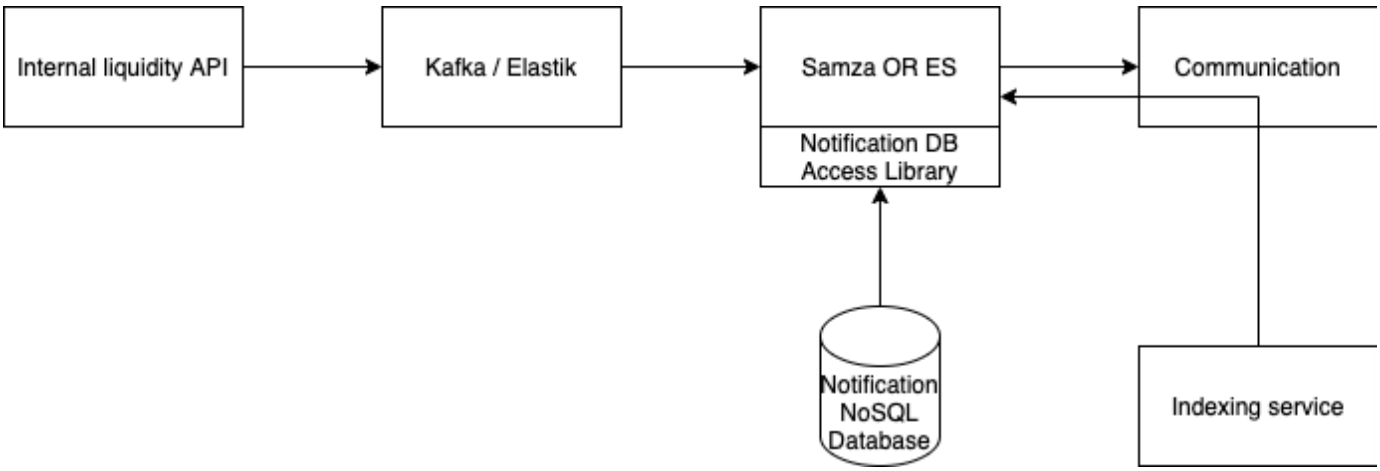
Implementation tip: Use AWS Data Pipeline, Spark, or similar services that can read from a message queue and process large amounts of data.

API response logs

When a user views a pool, both request and response are logged. Pool and coin/token price will be included in the logs.

One way to achieve this is to use Kafka or Elastic to stream the API request and response, then a Samza or ElasticSearch job will process the streamed API request and response.

With RDBMS, we can easily do a select statement based on the search query to match the rows in your notifications database.



NoSQL solution needs an Indexing service

Technical notes

- Services communicate via HTTPS calls.
- Redundancy is not covered in detail here. You can easily add a load balancer in front of the services that make up your Notification Service and run them in multiple regions.