Record-to-Record Synchronization Service: Design Document

Design a bi-directional record synchronization service that can handle CRUD (Create, Read, Update, Delete) operations between two systems:

- System A (Internal): We have full access, including back-end services and storage.
- System B (External): Accessible only via API. External APIs may have rate limits, and the data models may differ from our internal system.

Design Requirement

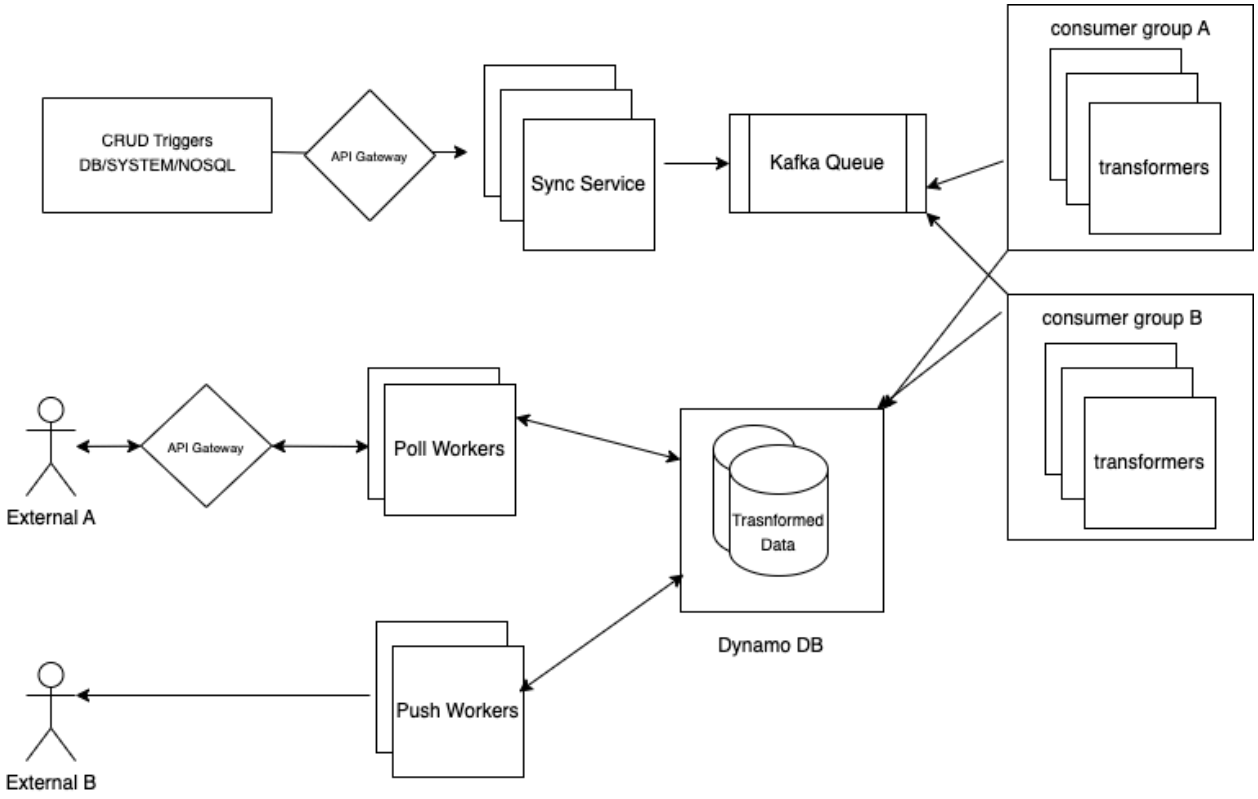
- Record by Record synchronisation request reaching 300M syncs/Day.
- Support multiple CRM providers
- Data transformation to match Destination system
- Validation of schema
- Sync actions (CRUD) are determined by pre-configured rules or triggers.

Subsystem Chosen Implementation : Syncing updates from internal to external servers

Assumptions Made:

1. Any CRUD operation performed on Database/System will have triggers configured to them and make requests to our sync service, here we have implemented for on record basis, but can be extended for batch as well.
2. The structure of the input data and customer data format are known in advance. And do not support dynamic change to the record data format.
3. The system currently doesn't support custom fields and columns. Will discuss systems to incorporate them in later suggestions.
4. Will also discuss the other flow system, and assumptions in later sections.

System Diagram



Important Components

SYNC SERVICE

This service will act as the entry point for our A->B sync service. Any CRUD operation performed on our postgres or DynamoDB will have triggers configured with it. Which in turn will hit our Sync service REST API. Since we are expecting 300M sync requests a day. We will have to introduce an API Gateway for load balancing and metric collection.

Key Points :

- Sync Tuple data with metadata is less than 1MB. The requirement also states record-by-record syncs.
- The schema of the internal data and external data are known in advance, not dynamic in nature.
- There is a schema folder which contains internal data, schema and is used to validate internal Data in the sync service entry point.
- Once Data is validated the data is pushed into the kafka stream for transformation workers.

SCHEMA VALIDATOR

The schema directory in the system has folders for each customer and internal data, containing the schema expected by each system. This validator can be reused for B->A service as well. In our System, internal data validation happens at Sync service. And external Data validation is performed by Transform workers and transforming data and before writing to Ingestion Queue/DB.

KAFKA QUEUE

All the CRUD events are stored in this queue, each data type is associated with a separate topic. Each customer is associated with a consumer group and can subscribe to only topics necessary and consume only those topics.

This currently has only one consumer group per customer, but can be extended to a consumer group per topic for each customer for independent ingestion of data and fault tolerance.

Transformer Service and Workers

These are essentially consumer services, which consume the kafka data and transform data to the External service format using Transformer service.

This is currently implemented using Factory pattern, with a Transformer Class for each Customer providing flexibility and easier extension.

Transformed Data Storage

The current implementation uses a dynamo db style database for storing the transformed data. This works really well for a polling style external service. Which provides a great degree of control to the external service, allowing them to poll using offset and limit.

But I would also like to suggest a hybrid approach, where we could also have a kafka style queue which allows us to push data to the external service in a real time manner. The rate-limit, retry logic are not implemented, but will ideally be handled here.

Dynamo DB, will provide support in case of retries as it could also act DLQ.

We will also have workers to push data, and serve requests to API requests in polling type updates.

Key Decision and Trade-Off Discussion

Kafka vs SQS

- Kafka was preferred as we need multiple customer workers to process the same data. And allows better partitioning and replay options in case of error handling

Json based schema validation

- Allows us to clearly define boundaries for data. Can easily be stored in blob or DB later.

Transformer

- The current implementation uses a Factory pattern for implementing multiple transform rules for customers. This was chosen for easier implementation though provides proper boundary transform and extensibility.
- But a big drawback with this is it is not suitable for dynamically changing systems. In that case it is advisable to store the transform mapping in DB and fetch it and perform key replacement on values. This will support all types of data.
- Another key extension not supported is custom fields that may be passed, it is also advisable to have those stored in a DB and perform transformation. As those may be organisation specific. Having them handled by Factory pattern becomes very difficult.

Key Future Extension

- Metrics integration, and decision making based on those metrics, like increasing/decreasing workers based on kafka lag.
- Detailed error handling
- Re-Try logic, to try multiple times. In case of failure, and also make decisions like if we should stop processing and inform customers of downtime or skip.
- Offset tracking by transformer to recover from disaster.
- Implement rate-limitters for push service.

Customer based configs are stored in config.py for development purpose.
Unit testing written test/folder.

You can go through read.me file to run the service and see logs in `logs/polling_service.log`