Comment Service Design

Owner: @ayman, @joey
Joseph Orme

History

List major version changes, e.g.,

- 2019/02/21: initial design work, added TODOs (joey)
- 2019/02/27: added notes on database flavors (joey)

Summary

The Comment Service is designed to reflect and reproduce the API endpoints of the PHP Monolith for all comment related services on the Mercari application. The service will allow users standard GET/POST/PUT operations on comments, query for the number of comments for items, DELETE comments (for item sellers), and to hide comments or update them (for administrators)

Background

With the switch over to the microservice architecture, comments will need to be created to handle all comments in the system. Currently in PHP, there are multiple comment SQL database tables, handled by Comment, Contact, and Comment Summary services. We feel this is an opportunity to converge these services, allowing for one “master of all comments” to simplify the overlying microservice architecture

Goals

- Create logic that works the same way currently in PHP monolith
- Provide a simpler DB structure to condense and simplify comment data

Non-goals

- Degrade performance of comment services in Mercari

Useful Links

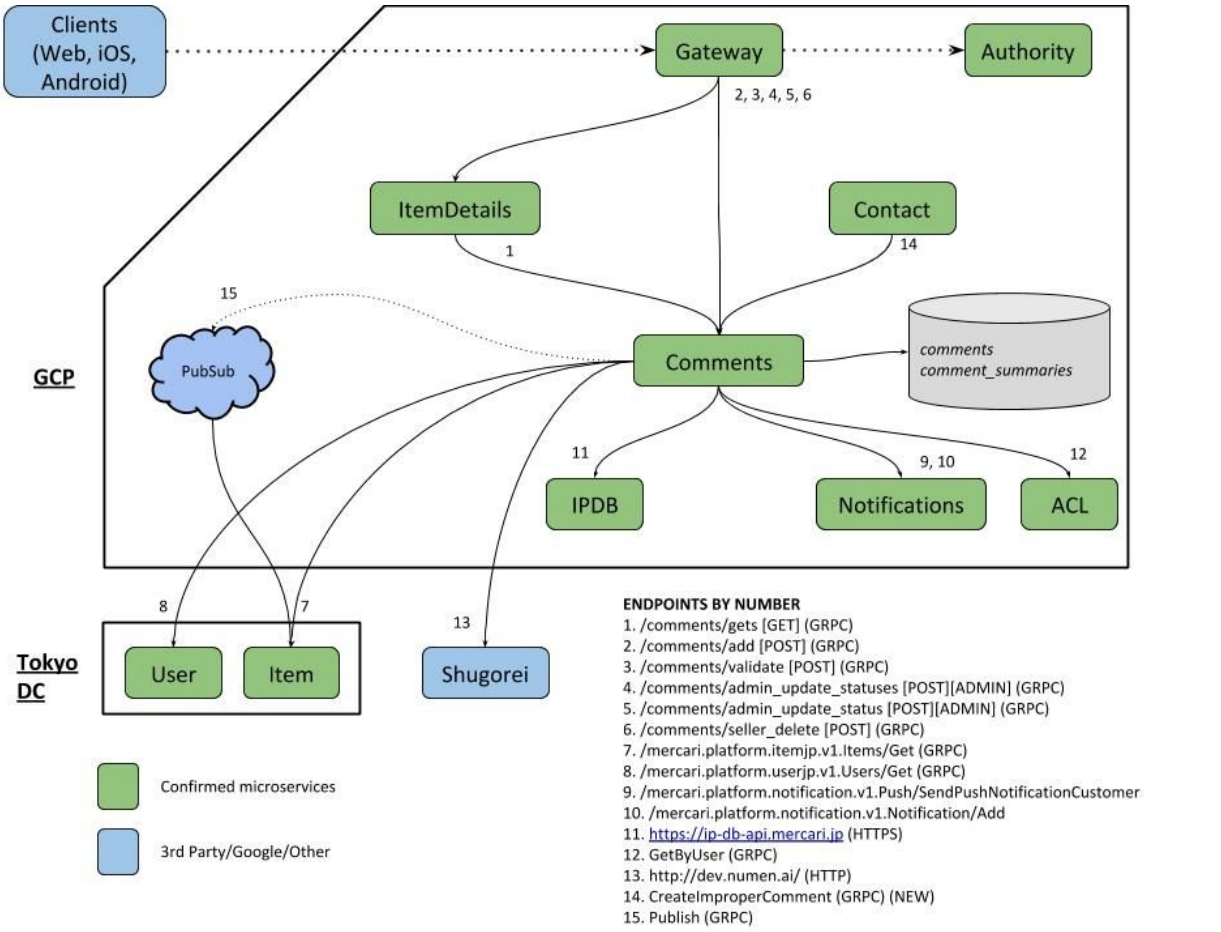
If any of these are available:

- Github repo
- Datadog dashboard

System Design

Architecture Diagram / Data Flow Diagram

Initial Architecture Goal (version x.x.x)



Ref: Comment Research

Caveats

- This is an ‘ideal’, meant to show the overall goal of the architecture of the comment space and how it would fit in regards to the overall application architecture. Migration to this idea will be covered in more detail in the **Data Migration** section of this document.
- Overall goals of the design is to keep compatibility with existing functionality, while migrating over to microservice architecture.
- The fetching of comments and the calculation of the number of comments for items are arguably the ‘hardest hit’ endpoints of the new service. The number of services that are needing this information are not shown for the sake of keeping the diagram clean and simple.

- Shugorei is a service hosted by Karakuri. This is used currently in PHP for comment validation for all newer versions of the client (older clients do not call this). We intend to enforce this validation when the comment microservice is called to add a comment to the system

Interfaces

- Access endpoints
    - Current HTTP endpoints:
        - /comments/gets [GET] (7 days avg: 667 request per second 'rps' for direct calls, and this is expected to be called when /items/get gets hit which will add more 2171 rps. In total ~ 2900 rps)
        - /comments/validate [POST] (7 days avg: 10 rps)
        - /comments/add [POST] (7 days avg: 14 rps)
        - /comments/seller_delete [POST] (7 days avg: 4 rps)
        - /comments/admin_update_status [POST] (7 days avg: ~ 1 rps)
        - /comments/admin_update_statuses [POST] (7 days avg: ~ 1 rps)

    - GRPC (mercari.platform.v1.Comments)(???):
        - ListComments
            - List the comments for a given item ID
            - ListCommentsRequest {
                `string` itemId = 1;
                // fields could be
                // fields = ['commentsNum'] => only the numbers
                // fields = ['commentsNum', 'comments.message']
                `repeated` fields = 2;
              }
            - ListCommentsResponse {
                `int32` commentsNum = 1;
                `repeated` comments = 2;
              }
        - GetComment
            - Get a specific comment by comment ID
            - GetCommentRequest {
                `string` commentId = 1;
                `repeated` fields = 2;
              }
            - Response => Comment
        - ValidateComment
            - Validating a comment
            - ValidateCommentRequest {
                `string` itemId = 1;

```
        string message = 2;
    }
```

- ValidateCommentResponse {

```
        bool validComment = 1;
    }
```

- CreateComment
  - Creating a new comment
  - CreateCommentRequest {

```
        string itemId = 1;
        string message = 2;
        repeated photos = 3;
    }
```

  - Response => Comment
- DeleteComment
  - Deleting a comment (Currently this delete endpoint “seller_delete” in Mercari API is allowing the seller only to deleting the comments on his items, and the code for that is written explicitly in the CommentsService. But, technically this is not the best approach to do things. Assume that we wanted to let the user who commented to go and to delete his own comment a code change will be needed. Technically a better approach is to “fine grain the scopes” in the PAT token so that it include the “privileges” of this user on different resources, then the services will not depend on a hard coded stuff to check if this user is allowed to delete or not. This kind of big change, but for now and for the sake of improving and get things done, the gRPC API will has the name of “DeleteComment” without “Seller” but the logic inside will do the same thing with a comment explaining the previous point, then when things changes we can just change the logic and keep the endpoints and interfaces as it’s.)
  - DeleteCommentRequest {

```
        string commentId = 1;
    }
```

  - Response => Comment
- FlagImproperComment
  - New endpoint, allows microservices to flag a comment as ‘improper’ and hide it from the system
- UpdateCommentStatus
  - Allows an admin to update the status of a comment for the given commnet ID
  - UpdateCommentStatusRequest {

```
        string itemId = 1;
        string commentId = 1;
        string status;
```

}
- Response => Comment

- BatchUpdateCommentStatus
  - Bulk endpoint for admins to update the status of the comments with the passed IDs
  - BatchUpdateCommentStatusRequest {
    `repeated` commentIds = 1;
    `string` status = 2;
    }
  - BatchUpdateCommentStatusResponse {
    `int` updatedCommentsNum = 1;
    }
- UpdateCommentImproperStatus
  - Add/Update an improper comment status, this endpoint is going to be called from “Contact MS”.
  - UpdateCommentImproperStatusRequest {
    `string` commentId = 1;
    `string` remoteAddress = 2;
    `optional string` ivCert = 3;
    `optional string` userId = 4;
    }
  - Response => Comment

Events

- Service publish or subscribe to pubsub:
  - Comments performs a mute publish to **PubSubService** for the following scenarios
    - When a comment is added to the system
    - When a seller deletes a comment

Batch Jobs

- Batch jobs:
  - Comment Reply Reminder
    - Notify the seller if they have yet to reply to a comment on their item in a given time period
  - Liked Item Received Comment
    - Notify users that an item they liked received a new comment

Expected Clients

The following list shows the clients that are expected to use the service, how they will use it, what kind of SLOs they will require, and roughly what kind of load they will add to the APIs. e.g.,

- Item-details
    - Calls ListComments and GetComment
    - Expecting a high load and traffic directed via the item-details service
        - /items/get is one of the hardest hit endpoints in the entire application with over 1.4k rpms a day
- Contact
    - Will call UpdateCommentImproperStatus
    - (02/22) This microservice is scheduled to start development next quarter
- Likes
    - Calls ListComments with fields = [‘commentsNum’] => only the number, basically for the purposes of adding event logs
    - (02/22) Still in development
- Transactions
    - Calls ListComments with fields = [‘commentsNum’] => only the number, basically for the purpose of adding event logs
- Gateway
    - All other traffic regarding comments (Add Comment, Update Comment, Admin Update)

Dependencies

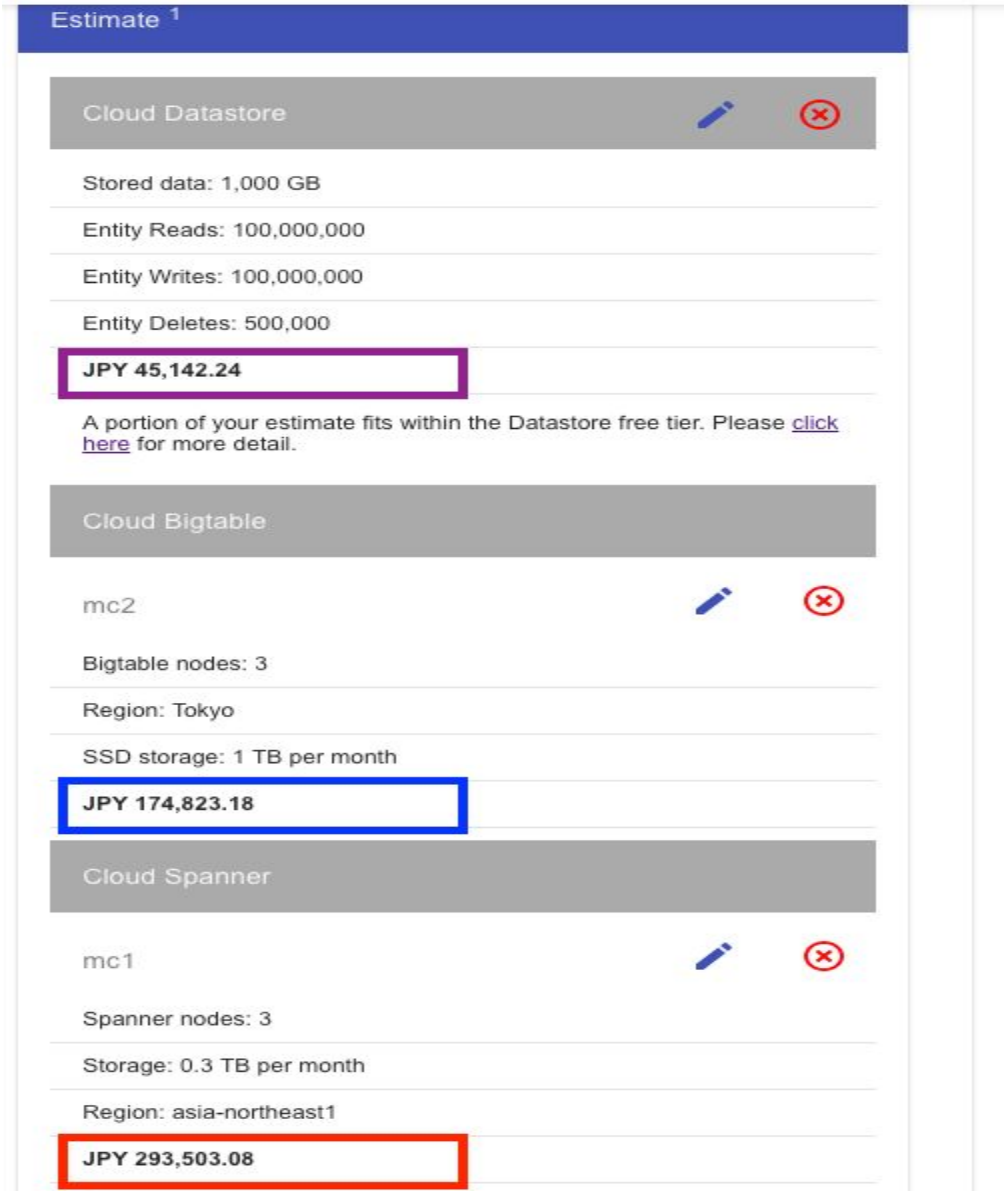
The following list shows the services (including 3rd party services, database tables or pubsub) that “Comments” service will depend upon. Also, it includes information about which dependencies are critical (i.e. they are potential causes of SLO violations for your service).

- user-service
    - Will call user-service to get user information, primarily the user IDs
    - Failure to call this service will result in being unable to associate a comment with a user, as well as validation for comments via Shugorei, ACL services
- item-service
    - Will call item service to get item data, primarily the seller ID and the item ID
    - Failure to call this will result in being unable to associate a comment with an item, as well as conduct proper notifications and batch jobs
- acl- service
    - Will call to acl service to validate the user’s ability to comment
    - Failure to do this will result in a comment validation error

SLO (Optional)

Current measurements for the services are as follows (via New Relic)

- comments/gets avg. ms: 7ms
- comments/gets is 400rpm/day Database



- **DataStore**
    - **Features**
        - Scales well
        - Transactions work on this!
        - Price based on Storage Capacity and Reads/Writes

                - **100M Reads/Writes for 1000TB of storage = 50k/Month**
                    - Scales if you over/under use
            - Indexing and Range Scans
            - **No Concept of Full Scan**
                - **Query fails if you can't find a huge data set**
                - **Makes complex queries and aggregations hard/impossible**
                - Limited to simple queries on simple indexes/keys (ascending/descending)
            - Eventual consistency
                - **Stale data possible**
                - Some MSs might be okay with this
            - Entity Group
                - Child/Parent Relationships for Data
            - Ancestor Queries
                - Grabs all children of given entity
                - Strong consistency
- **BigTable**
    - **Features**
        - Distributed Keys: used to expand storage fast and easy
        - Key strategy required
            - Hashes
            - Ordering
        - Simple K/V Store
        - Keys link together for more complex
        - **NO TRANSACTION CAPABILITY (Except per row)**
        - **60k/Month, 3 Tables recommended = 180k/Month**
        - Automatic scaling/distribution depending on data nature (tablets increase/decrease/rearrange based on data)
- **Spanner**
    - **Features**
        - **SQL DB**
            - Strong consistency
            - Strong Transactional Processes on Indexes
        - **Transactional DB!**
        - **Complex Queries!**
        - Key Strategy Required to avoid hot spots
            - Works similar to Datastore/Bigtable with keys and distribution
            - UUID4 is recommended by Google per the Documentation
            - Sharding
            - Hashes
        - **EXPENSIVE $$$$$**
            - **Recommended setup of 3 nodes at 2TB each = 270k**
        - Interleave

- Similar to joins, only with strong data relationships
    - Parent/child similar to DataStore
    - Keeps the Data in the same place, for faster reads

New Tables

If new tables will be created, will they be replicated to BigQuery? If so, how?
In most cases, it is necessary to replicate the data to BigQuery so that product / BI team can use it for analysis. For the databases in Sakura, SRE has already set up the ETL process so developers don't need to worry about it.

Changes to Existing Tables

We plan to change the schema, to reduce complexity and redundancy of data. The following tables work in either a NoSQL or SQL flavor of database

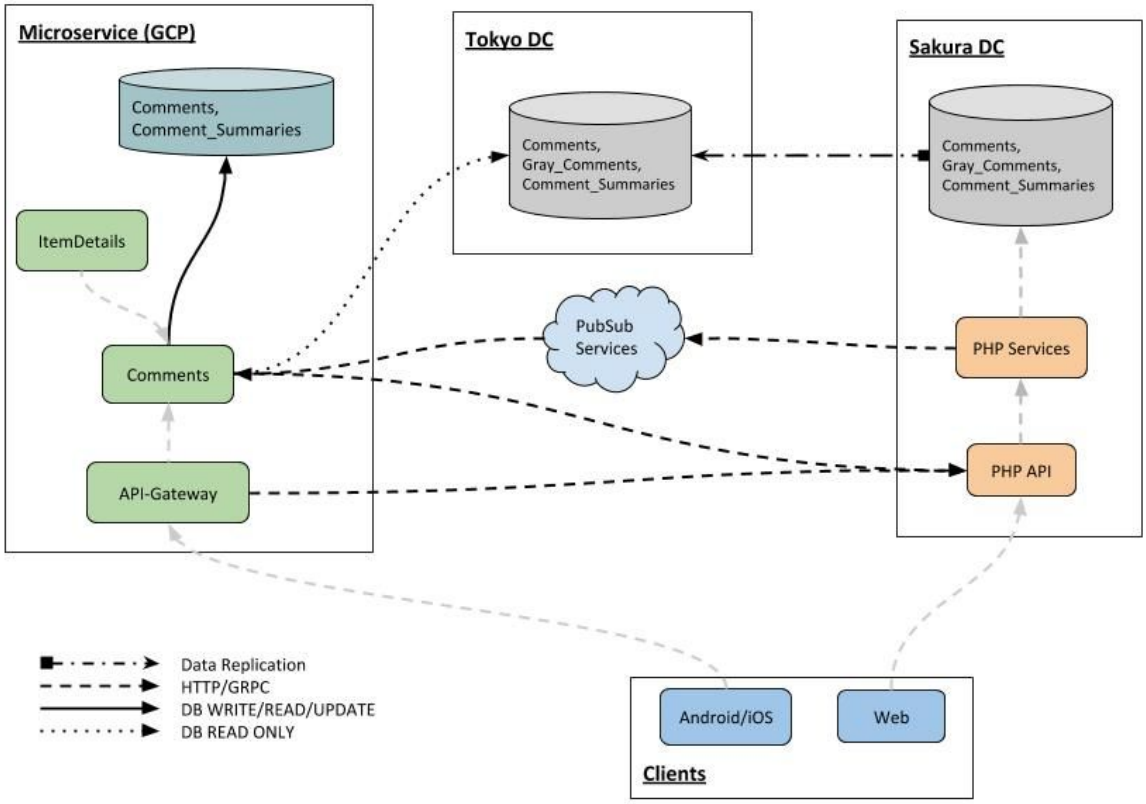
```
Comment {
        Comment_ID (Key)
        User_ID (Key)
        Message (String)
        Status (Enum)
        Grey? (Bool)
        Improper? (Bool)
        Created (timestamp)
        Updated (timestamp)
}

Comment_Summary {
        Item_ID {
                Comment_IDs[...]
        }
}
```

For more information on migration strategy refer to the **Database Migration** section of this document

Database Migration

Data Migration Strategy Diagram



Caveats and Considerations

- To keep compatibility with the existing tables, we initially call to the PHP API to add/update/delete comments before the existing logic is moved over. This will land the data in the tables in Sakura DC, which eventually will be replicated to Tokyo DC
    - This is an example of the Strangler Pattern
- There are a few strategies to handle incoming comments once enough service logic has been successfully transferred to the comment microservice
    - 1) Do a 'mass' migration of data from the Tokyo DC to the new tables for comments. One time only and then ensures that all data is inside the new tables. We can only do this once the service is ready to be cut off entirely from PHP services.
    - 2) Do a 'gradual' migration of data, taking advantage of existing PubSub service calls made when a new comment is made to the system, or when a comment is updated. Comments microservice could subscribe to this published event, and slowly move over data to fill the table out. This allows us to leverage the PHP services longer, until we are ready to direct all traffic to the microservice.
- Web traffic will eventually be moved to call the API Gateway instead of the PHP directly. When that happens, it should be completely safe to not call the PHP APIs anymore once all APIs have been successfully added to the comment microservice

PII Considerations

There is no personal information kept in comments. The only information kept is the comment ID, associated item ID, and the associated user ID.

Security Considerations

E.g.,

- All requests should provide a valid PAT, validated by authority-service

Alternatives (Optional)

If you have considered other solutions, mention them here, along with their pros, cons, and the reason they were discarded.

New Technologies Introduced (Optional)

If the service introduces a new technology that hasn't been widely used in Mercari, e.g., MongoDB, introduce it and explain why it's necessary.

References

If any
Google Price Estimates