System Design Document

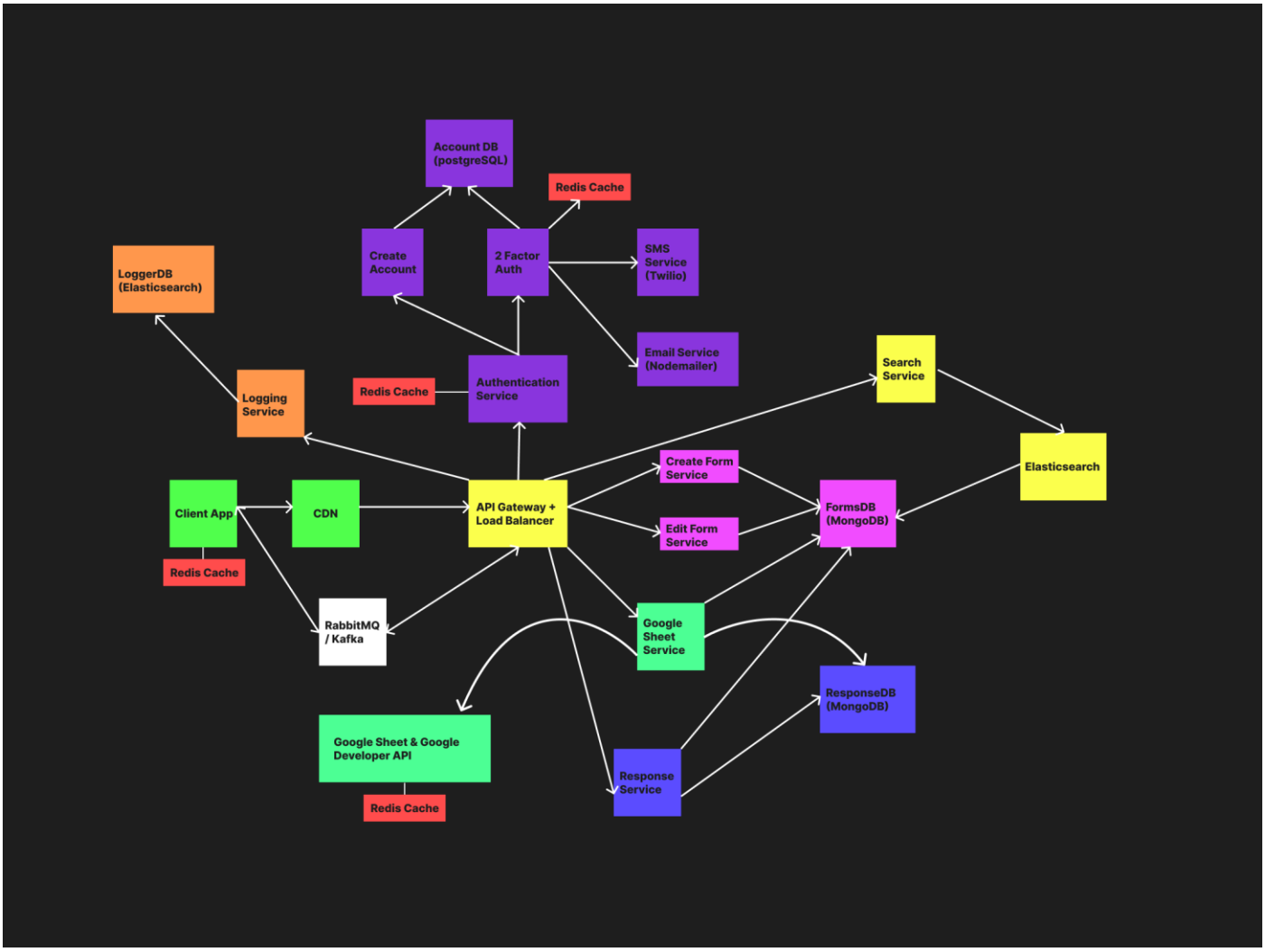
For

Google Forms Clone backend

By:

Name: Apoorv Bedmutha

MTech (Computer Networks) @ IIIT Gwalior
Contact: +91 9561295237
bedmuthaapoorv@gmail.com



System Design

Overview:

- Data Collection Platform (similar to google forms)
- Geographically Scalable (multiple countries)
- 250+ organizations (i.e many creator accounts)
- 11 million responses (heavy traffic)

Features:

- Offline Data collection
- Post processing of forms
- Customizable forms
- Searching capablities
- Exporting to google sheeets
- SMS Notification system
- Logging

Functional requirements:

- Customizable
- Easy to add and detach services
- Eventual Consistency
- Fault tolerant
- Google sheet API Limitation must be answered

Basic Architecture:

- Since there will be a huge variety of consumers, there will also be a huge diversity between the devices used to respond the forms. Possible options for creating the client application are:
    - **Mobile App -> OK**
    - **Desktop App -> OK**
    - Website -> NO, won't be ideal as we also need to do offline data collection

  Hence, the **client application will be either a mobile App or a desktop app or both**.

- The architecture must be Plug & Play, ie. Addition & removal of new & old services must be easy. To address these requirements we must adhere to the **Microservices architecture.**

Since, We want the system to be **fault tolerant** to **inconsistent bandwidth & facilitate offline data collection**.

we will make use of a message queue for storing the response to be sent until bandwidth is availaible.

Also Instead of repeatedly requesting the same form multiple times,we can store it in a cache.

Options for message queue:

- RabbitMQ: NO, as we may also want to run analytics or processing on the responses, which will not be provided by RabbitMQ
- **Kafka** -> OK

Options for Cache:

- Memcached-> NO, is difficult to scale to an enterprise application
- **Redis** -> OK, can be easily upgraded to enterprise use cases

Therefore client App will have:

- **MQ: Kafka**
- **Cache: Redis**

All the CSS & Images that the application needs can be stored in a **CDN such as AWS Cloudfront** near the user's geographical location.

In order to implement **Microservices architecture**, we will be needing multiple servers and for proper channeling of requests we'll need an **API Gateway & a load balancer such as Elastic Load Balancer(ELB).**

Types of profiles a user can create:

- Consumer : can only respond to a form
- Creator : can create, edit and respond forms

In order to uniquely identify each user, we need to implement **authentication**. Below are the options:

- Username-Password -> NO, Users will create a lot of accounts, in efficient, less secure
- Biometric -> NO, will break the cross platform needs
- **Two Factor Authentication (2FA) -> Perfect, easy to use, More secure than username-password**
- Passwordless authentication -> NO, Most secure but will bring in unecessary complexity as the use of our App will be like a survey & surveys will only work if it is easy for the consumer to participate in it

Hence, for our case 2FA is perfect, to achieve this we will need some form of SMS / email service which will send OTP to the consumer.

- **SMS service: Twilio**
- **Email service: Mailchimp**

It is very much possible an authentication server will receive redundant authentication requests, to **avoid reprocessing** & **minimize the risk of a DOS attack**, it's best to introduce a **redis cache that will store the response for a request for a pre-defined period of time.**

The details of an user account must be stored in a relational DB as it's structure is not supposed to change frequently, also relation Databases are faster for retrieval of data.

Possible options for relational DB:

- MySQL -> NO, only good for testing purposes, not efficient enough for enterprise applications
- **PostgreSQL -> OK, PostgreSQL performs better than MySQL in terms of indexing, data types, ACID compliance, concurrency and read & write performance.**

Schema of User Account Table:

- Username -> Primary Key, varchar
- Password -> Not null, varchar
- Contact -> Not null, int(10)
- Email -> Not null, must follow email format,
- Account type -> int (0: consumer, 1: creator)

Eventual Consistency:

This comes into play when we need to edit an existing form, any change made to a form needs to be populated gradually to all concerned devices. But the devices might or might not be connected to the network at the moment. Hence making use of the Message Queue will help creating a distributed system of forms.

There are different tehciniques via which this is possible:

- Master-slave: NO, has a single point of failure if a master crashes.
- **Multi-master-slave: OK, More fault tolerant than master-slave**
- Active-active: NO, Not necessarily needed as not all nodes need to have the complete forms DB, as discussed earlier each client app will cache only the forms it needs., Hence making all devices active introduces unnecessary complexity.

Therefore, It's best to go forward with Multi-master slave approach.

Steps of Eventual consistency:

- Client App will edit the form
- This new form will be stored locally in redis
- The changes (only the changes) will then be sent to Kafka
- All the master servers will be subscribed to the client's Kafka
- As soon as it is possible they will consume the change from cache & update the formsDB
- Also they will populate this change in their own Kafka, from which the rest of the client device's will grab the changes & update their copy of the form.

In this manner, we can achieve eventual consistency.

Forms are customizable and the questions each form pocesses can be of differenct structure. **Such forms will undergo many insertions and updations hence it'll be efficient to store them in a NoSQL DB such as MongoDB rather than a relational one.**

Structure of forms collection:

```
[
    {
        formID:<unique form id>,
        questions:[
            {
                Qcode:<unique question code>,
                Text: <question text>,
                Type:<number representing type of question>,
                Options(optional):[<option1>, <option2>,…]
            }, ….
        ],
        metadata:{
            author:<username of form creator>,
            timestamp:<time at which the form was last updated>
        }
    }, …..
]
```

Example of Question type code:

0: subjective question -> expects text output

1: MCQ

2: MSQ

And so on.

Types of modifications possible by the user:

1: add a new question

2: delete an existing question

3: modify an existing question

Structure of Change message:

```
{
    ActionCode: <action code>,
    FormID: <form id>,
    Qcode: <qcode>,
    Question (not needed for deleting a question):{
        <the new question, structure same as the forms>
    }
}
```

For cases when conflicting modifications are received the one with latest time stamp will be carried out.

Storage of responses:

The actual responses will be stored in mongodb due to the possibility of different structures

Each response will be assigned a unique responseID

The mapping of this responseID with it's corresponding formID & responderID(i.e. username of responder) will form a fixed structure hence will be stored in postgreSQL

Structure of Responder Table:

- responseID -> primary key
- responderID -> not null
- formID -> not null, index

formID will be indexed because we need to export the responses regarding a form to google sheets, for which searching a form must be easy.

Structure of Responses collection:

```
[
    {
        responseID: <responseID>,
        responses:[<response1>, <response2>, …]
    }
]
```

Further if we need to run search on the response itself, **it'll be great to introduce a search engine DB like elasticsearch.**

Limitation of Google Sheets API:

The Google Sheets API, allows us to create, delete and edit a google sheet through code, but the limitations include:

- read / write request limit of 300 per minute per project
- read / write requests per minute per user per project: 60
- If limit is crossed, user has to wait for back off time to finish before new request can be sent.

This limit is very much possible to reach with a large no. of users, below are some optimizations:

- implement cache at the GoggleSheetsServiceLayer, so that **redundant requests will be cached**
- **Do not allow more than 60 requests in a minute, hold rest of the requests in a buffer/queue and consume the next 60 requests in next minute.**

To ensure that each service is actually isolated and guarantee cross platform application, we must containerize the application using Docker

APIs:

```
    app.route('/').get((req, res)=>{
        res.send('App is working')
    });
    // for creation of a new account
    app.route('/createAccountService').post(controller.authenticationService.createAccount
Service);

    // 2 factor authentication
    app.route('/twoFAService').post(controller.authenticationService.twoFAService);

    // 2 factor authentication
    app.route('/validateOTPService').post(controller.authenticationService.validateOTPServ
ice);

    // create form service
    app.route('/createFormService').post(controller.formService.createFormService);

    // edit form service
    app.route('/editFormService').get(controller.formService.editFormService);

    // // post process logic response
    // app.route('/postProcessFormService').get(controller.postProcessFormService);

    // get form
    app.route('/getFormService').post(controller.formService.getFormService);

    // response storage
    app.route('/responseStorageService').post(controller.responseService.responseStorageSe
rvice);

    // // logging service
    // app.route('/loggingService').get(controller.loggingService);

    // google sheet generation service
    app.route('/googleSheetService').post(controller.googleSheetService.exportToSheetServi
ce);
```

Details:

- Tech stack used:
    - NodeJS: for building a backend project
    - ExpressJS: for building a Server & hosting APIs
    - Redis: for Caching and as a in memory DB
    - Kafka: as MQ
    - PostgreSQL: for relational DB
    - MongoDB: for distributed NoSQL DB
    - cURL & Postman: for API Testing
    - Google Sheets API
    - Docker: for containerizing the applications

PS: I have followed the **MVC Architectural Pattern & Data Access Object Design Pattern** for proper organization and easy detection of bugs.