Forms

Approach

- Decided to store the questions in the form of doubly linked-list where each DB entry will store the previous question id and next question id.
  This was done to ensure that the questions are displayed in order in the form specially after several insertions and deletions of questions.
  - Another approach could have been to use decimal numbers with each question and display them in increasing order of this number. When a new question would be added its display number would have been the average of the display numbers of the preceding and next question. This method was disregarded as the precision of decimal values would increase as the number of insertions increases. Also I felt this way is very informal.
  - Another approach was using singly linked list storing just previous question id. While this is less space intensive but for deletion of a question it would require the id of the next question(for updating the linked list). If the size of DB has to be reduced, this approach works better
  - In the Google Sheets, the data is therefore displayed in order.
- Bifurcated the answers into two tables, one storing the mcq answers and the other storing text answers as the text answers could be very long(~VARCHAR(200)) and initializing such values for MCQ type questions does not make sense whose valid responses is stored in response table.
- Stored responses to MCQ questions this helps in storing just response_id in answer_mcq table which reduces space.(As we moved from VARCHAR(~50) to INT datatype)
- Implemented atomicity in order to ensure atomic operations for all API calls. This ensures that the if the system fails at any point where the data was not saved the entire entry is rolledback and consistency is ensured.
- Here are some of the **limitations** of the Google Sheets API:
  - **Quota limits**: The Sheets API has per-minute quotas, and they're refilled every minute. For example, there's a read request limit of 300 per minute per project. If your app sends 350 requests in one minute, the additional 50 requests exceed the quota and generates a 429: Too many requests HTTP

status code response.
This can be solved using exponential backoff.
Exponential backoff is a standard error handling strategy for network applications. An exponential backoff algorithm retries requests using exponentially increasing wait times between requests, up to a maximum backoff time. If requests are still unsuccessful, it's important that the delays between requests increase over time until the request is successful.

- **Maximum payload size**: While the Sheets API has no hard size limits for an API request, users might experience limits from different processing components not controlled by Sheets. This should not be an issue for our usecase

DB Creation

Requirements :

1. User creates forms
2. Forms have questions
3. Questions have Responses
4. User creates answers
5. Each answer relates to a question

Entity Set:

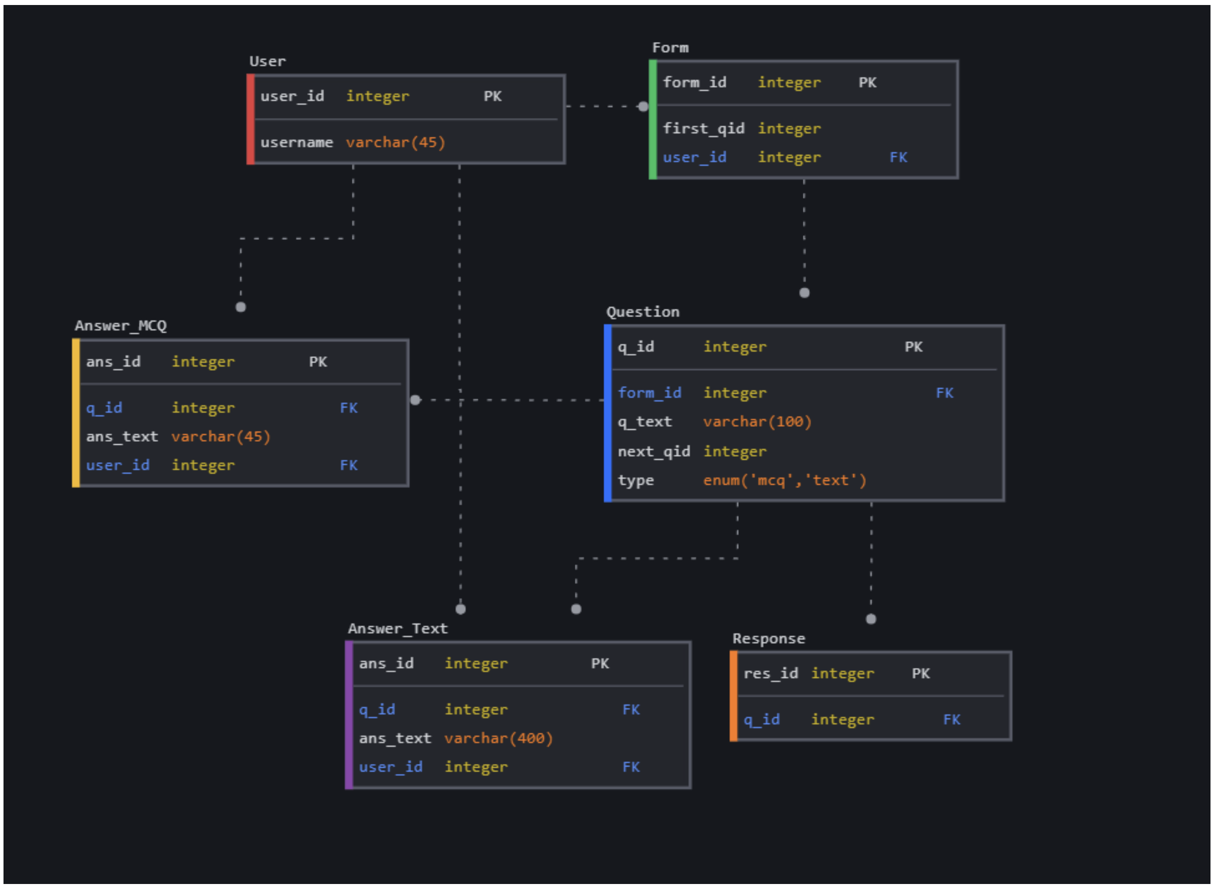
1. User
2. Form
3. Question
4. Response
5. Answer_Text
6. Answer_MCQ

Attribute Set :

1. User - **user_id** , username
2. Form - **form_id**, user_id, first_qid
3. Question - **q_id**, form_id, q_text, next_qid
4. Response - **res_id**, q_id, res_text
5. Answer_Text - **ans_id**, q_id, user_id, ans_text
6. Answer_MCQ - **ans_id**, q_id, user_id, ans_text

Relations:

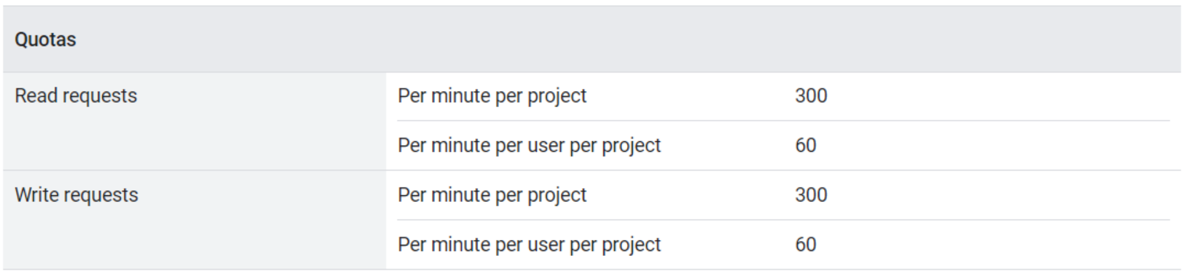
1. User creates forms
   - 1:N, forms is weak entity, user is the strong entity
2. Form has questions
   - 1:N, forms is strong entity, question is the weak entity
3. Question has Responses
   - 1:N, response is weak entity, question is the strong entity
4. User creates answers
   - 1:N, answer is weak entity, user is the strong entity
5. Each answer_text relates to a question
   - 1:N, answer_text is weak entity, question is the strong entity
6. Each answer_mcq relates to a question
   - 1:N, answer_mcq is weak entity, question is the strong entity



```
CREATE SCHEMA IF NOT EXISTS `form_manager` DEFAULT CHARACTER 
USE `form_manager`;

CREATE TABLE IF NOT EXISTS `user` (
  `user_id` INT NOT NULL AUTO_INCREMENT,
  `username` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`user_id`)
) ENGINE=InnoDB DEFAULT CHARACTER SET=utf8mb4 COLLATE=utf8mb4_

CREATE TABLE IF NOT EXISTS `form` (
  `form_id` INT NOT NULL AUTO_INCREMENT,
  `first_qid` INT NULL DEFAULT NULL,
  `user_id` INT NULL DEFAULT NULL,
  `spreadsheetId` VARCHAR(45) NULL DEFAULT NULL,
  PRIMARY KEY (`form_id`),
  INDEX `user_id_idx` (`user_id` ASC) VISIBLE,
  CONSTRAINT `user_id`
    FOREIGN KEY (`user_id`)
    REFERENCES `user` (`user_id`)
) ENGINE=InnoDB DEFAULT CHARACTER SET=utf8mb4 COLLATE=utf8mb4_

CREATE TABLE IF NOT EXISTS `question` (
  `q_id` INT NOT NULL AUTO_INCREMENT,
  `form_id` INT NOT NULL,
  `ques_text` VARCHAR(45) NULL DEFAULT NULL,
  `ques_type` ENUM('mcq', 'text') NOT NULL,
  `next_qid` INT NULL DEFAULT NULL,
  `prev_qid` INT NULL DEFAULT NULL,
  PRIMARY KEY (`q_id`),
  INDEX `form_id` (`form_id` ASC) VISIBLE,
  CONSTRAINT `question_ibfk_1`
    FOREIGN KEY (`form_id`)
    REFERENCES `form` (`form_id`)
) ENGINE=InnoDB DEFAULT CHARACTER SET=utf8mb4 COLLATE=utf8mb4_

CREATE TABLE IF NOT EXISTS `answer_mcq` (
  `ans_id` INT NOT NULL AUTO_INCREMENT,
  `q_id` INT NOT NULL,
  `res_id` INT NOT NULL,
  `user_id` INT NOT NULL,
  PRIMARY KEY (`ans_id`),
  INDEX `user_id` (`user_id` ASC) VISIBLE,
  INDEX `answer_mcq_ibfk_1` (`q_id` ASC) VISIBLE,
  CONSTRAINT `answer_mcq_ibfk_1`
    FOREIGN KEY (`q_id`)
    REFERENCES `question` (`q_id`),
  CONSTRAINT `answer_mcq_ibfk_2`
    FOREIGN KEY (`user_id`)
    REFERENCES `user` (`user_id`)
) ENGINE=InnoDB DEFAULT CHARACTER SET=utf8mb4 COLLATE=utf8mb4_

CREATE TABLE IF NOT EXISTS `answer_text` (
  `ans_id` INT NOT NULL AUTO_INCREMENT,
  `q_id` INT NOT NULL,
  `ans_text` VARCHAR(400) NOT NULL,
  `user_id` INT NOT NULL,
  PRIMARY KEY (`ans_id`),
  INDEX `user_id` (`user_id` ASC) VISIBLE,
  INDEX `answer_text_ibfk_1` (`q_id` ASC) VISIBLE,
  CONSTRAINT `answer_text_ibfk_1`
    FOREIGN KEY (`q_id`)
    REFERENCES `question` (`q_id`),
  CONSTRAINT `answer_text_ibfk_2`
    FOREIGN KEY (`user_id`)
    REFERENCES `user` (`user_id`)
) ENGINE=InnoDB DEFAULT CHARACTER SET=utf8mb4 COLLATE=utf8mb4_

CREATE TABLE IF NOT EXISTS `response` (
  `res_id` INT NOT NULL AUTO_INCREMENT,
  `q_id` INT NOT NULL,
  `res_text` VARCHAR(45) NULL DEFAULT NULL,
  PRIMARY KEY (`res_id`),
  CONSTRAINT `fk_q_id` FOREIGN KEY (`q_id`) REFERENCES `quest
) ENGINE=InnoDB DEFAULT CHARACTER SET=utf8mb4 COLLATE=utf8mb4_
```

Google Sheet Insertion:

- Made the google sheet insertions asynchronous.
- This improves the response time of the APIs as we dont have too wait for sheet insertion during API call.
- Google sheets has limit on requests per minute.



Quotas		
Read requests	Per minute per project	300
	Per minute per user per project	60
Write requests	Per minute per project	300
	Per minute per user per project	60

- If the limit is exceeded or the API has a lot of load we need to retry after giving the API some time. Thus I implemented the exponential backoff algorithm which delays the request for exponentially increasing time duration.

- I implemented the delay to end after the 32 second mark. This is because the total delay uptill this point is more than 60s. This would put the request in the next batch of requests(in the next minute).

Logging Service:

- Implemented the logging service in asynchronous fashion.
- This was done to accommodate the metrics calculation service. The API waits for the logs but not for the metrics calculation function.
- Implemented a rotating queue which retains the requests in the last 1 minute only. This way we can calculate the fraction of statuses dispatched in last minute.

Monitoring Service:

1. API Endpoint Monitoring:

- **Ping Tests:** Regularly check the API endpoints by sending periodic requests to ensure they're responsive.
- **HTTP Status Code Monitoring:** Monitor HTTP response codes to identify any errors (e.g., 4xx, 5xx codes). This can be accomplished through monitoring logs.
- **Latency Measurement:** Measure response times to ensure they are within acceptable thresholds.

2. Error Tracking:

- **Error Logs:** Monitor and analyze error logs to identify and troubleshoot issues.
- **Error Rate Monitoring:** Track the percentage of failed requests or errors over time.

3. Performance Metrics(Benchmarking):

- **Throughput:** Monitor the number of requests processed per second or minute.
- **Response Time:** Measure the time taken by the API to respond to requests.
- **Traffic Volume:** Analyze the amount of data transferred or received by the API.
- These can be measured by running stress tests on the API to get an idea on the performance of the API.