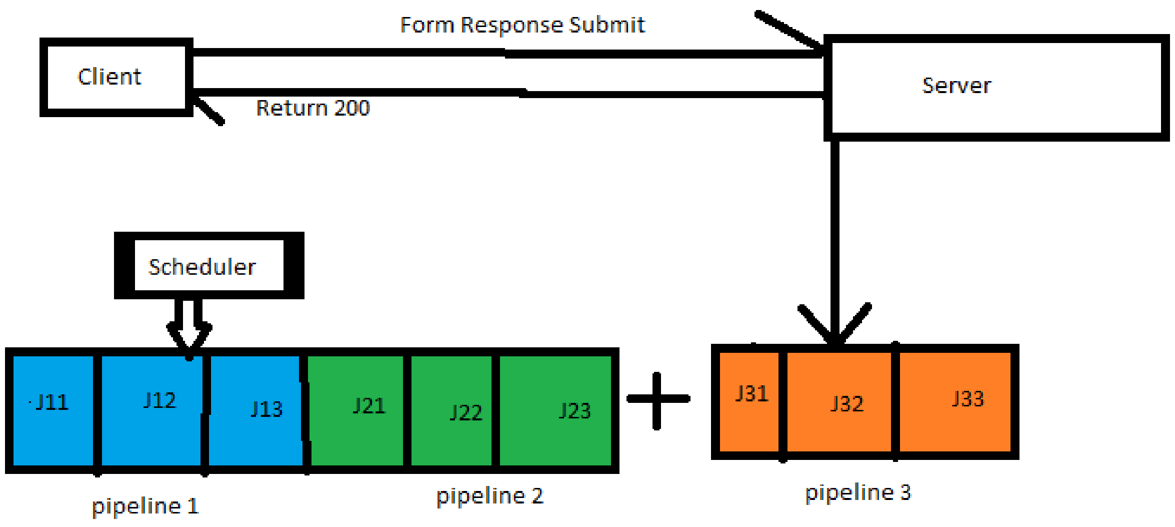
Design:

Below is the figure describing the workflow for the post submission logic using the proposed approach **pipelines**.



Workflow:

- The client submits the form response.
- The server performs the form submit logic and in the end, if everything is a success, returns a response to the client and creates a pipeline in the background.
- This pipeline is in the queue with other pipelines and is executed by a background scheduler
- For simplicity pipelines are being executed with a background scheduler, there are other approaches that we can use to execute pipelines.
- Each pipeline consists of several jobs which in our case are the features of post submission logic and are executed one by one.
- Order of execution can be customized and is up to us.
- Pipelines are run based on the order of creation, that is, in a FIFO fashion.

Pipeline Implementation:

A Pipeline can be stored as a table in our database, so to add a pipeline all we need to do is to perform a single insert operation. The schema of the pipeline is as follows:

Pipeline:

- Id: unique Id of the pipeline that is used as a primary key.

- response_id: Id of the response which caused the pipeline.
- Created_at: Holds the DateTime of when the pipeline is created and is used to decide the order of execution of pipelines.
- Status: pending, running, finished
- Finished at

Jobs Implementation:

A job is also stored as a record in our database in the table jobs. We associate jobs with pipelines and foreign key constraints can be used to implement this.

Jobs:

- Id: Id of the job, Primary key.
- Pipeline_id: Id of the pipeline with which the job is associated.
- Created_at
- Status: pending, success, failure
- Trigger - specifies which function to execute when this job is being executed.

Scheduler Logic:

The pipelines can be executed in a queue-based architecture where every time we create a pipeline we queue it using an enqueue function and dequeue it once its execution is complete but since I am using scheduler to run there can be some edge cases where we ought to be careful.

Since we need to execute the post submission as soon as the response is submitted we need to run the scheduler very frequently. I am running the scheduler every 5 seconds.

Now consider the scenario where there is a service outage/power outage and the server stopped in the middle of an execution. Now the pipeline is marked as “running” but it got interrupted because of an outage. We can solve this by executing all the pipelines that are not marked as finished. But consider a case where currently a job scheduled by the scheduler is not finished and 5 seconds is up and a new job is scheduled; this causes a pipeline to be executed multiple times. Although many libraries such as apscheduler prevent this, there is no harm from being careful.

To avoid such cases I am limiting the number of pipelines a scheduler can execute in a single run to 5, giving each pipeline 1 second to execute. This number can be properly found from practical observations.

So when a scheduler starts executing a pipeline it marks it as “running” and if it is interrupted by a power outage, then it is simply executed in the next scheduler run.

Scheduler: Executes every 5 seconds and in every 5 seconds it executes 5 pipelines.

Tech Documentation:

Tech Stack:

- Python FastAPI Framework
- PostgreSQL Database
- SQLAlchemy ORM
- APScheduler

The models used for the implementation are:

Models:

- User
- Form
- Question
- Response
- Answer
- Pipeline (Schema declared above)
- Job (Schema declared above)

User:

- Id
- Username
- Password
- Email
- Mobile Number
- Created at

Form:

- Id
- Owner_id: Id of the user who created the form
- Title
- description
- Created at
- Updated at

Question:

- Id
- Form_id: Id of the form to which question belongs to
- Question: string containing the question.
- Is_required
- Created at
- Updated at

Response:

- Id
- Form_Id
- User mobile number - Mobile number of the user who is responding to the form.
- Created at

Answer:

- Id
- Response_id
- Created at
- Question Id
- Answer: string containing the answer to the question.

To keep the implementation simple and demonstrate the approach multiple types of answers such as integer, float, etc. answers are not included but can be implemented, Also various form features can be added, for example, a form that does not allow multiple submissions, a form that closes after a certain time. But to keep the implementation simple for demonstration these features are not implemented.

Forms are flexible, they are similar to google forms where we can add/update/delete questions.

API Endpoints:

User:

- Create user - /auth/register
    - Anyone can access
    - Payload:
        - Username - Body
        - Password - Body
        - Email - Body
        - Mobile Number - Body
- Login user: /auth/login
    - Anyone can access
    - Payload
        - Username - Body
        - Password - Body
- Get logged in user details: /auth/user
    - Only logged in users can access

Form

- Create form - /form POST

    - Only logged in users can access
    - Payload
        - Form title
        - Form description
- Get form details - /form/<form_id> GET
    - Anyone can access
- Update form properties - /form/<form_id> PATCH
    - Only logged in users who created the form can access
    - Payload
        - Title
        - Description
- Delete form - /form/<form_id> DELETE
    - Only logged in users who created the form can access

Question

- Create question - /form/<form_id>/question POST
    - Only logged in users who created the form with id=form_id can access
    - Payload
        - Question
        - is_required
- Get all questions of the form - /form/<form_id>/question GET
    - Any user can access
- Get question details - /form/<form_id>/question/<question_id> GET
    - Any user can access
- Update question - /form/<form_id>/question/<question_id> PATCH
    - Only logged in user who created the form can access
    - Payload
        - Question
        - is_required
- Delete question - /form/<form_id>/question/<question_id> DELETE
    - Only logged in user who created the form can access

Response

- Get all responses of a form - /form/<form_id>/response?limit=10 GET
    - Only logged in user who created form can access.
- Create response - /form/<form_id>/response POST
    - Any one can access.
    - Important endpoint where pipelines for post-submission logic are implemented.
    - Payload:
        - Dict with keys as question_id and value as the answer to that question.
        - Mobile Number of the user that submits the form response.

Answer

- Get all answers for a given question - /form/<form_id>/question/<question_id>/answer?limit-10 GET
    - Only logged in users who created the form can access

Pipeline analytics:

- Get pipeline details to **monitor** - /pipeline/<pipeline_id> GET
    - Logged-in users who created the form can access it.
    - Returns full details of the pipeline mentioned and its jobs.
    - Details include, the number of jobs, the status of the pipeline, successfully executed jobs, failed jobs, etc.
- Get all pipelines for a form - /pipeline/form/<form_id>?limit=10 GET

Authentication

Simple jwt authentication to be implemented. A token is needed to access restricted endpoints. Tokens have an expiry time of 1 hour.

Monitoring Pipelines

Endpoint /pipeline/<pipeline_id> can be used to fetch pipeline analytics and can be used to monitor the execution and logging can be implemented to record the pipeline summary, etc, and can be used to debug failures.

Post submission Features Implementation:

Google Sheets

- A google sheet is created and shared to the user's email when a form is created.
- Google sheets are updated with columns whenever the question is added/updated/deleted
- The whole sheet will be deleted when the form is deleted by the user.
- Responses are updated in google sheets in real time whenever a response is submitted.
- Responses will be added in google sheets when the job of trigger type "GOOGLE_SHEETS" is run by the scheduler when executing the pipelines.
- None of these operations gives an operational overhead to the API calls and are executed in the background asynchronously.
- One of the **Limitations** of the used implementation is the google sheets writing limit which is 60 operations in a minute. When a Large number of responses are received and there are a lot of features that need to be executed this number can be exceeded. One way to prevent this is by controlling how many pipelines a scheduler should run in a minute.

Send SMS for receipt

- Print the SMS content and save it log.
- SMS implementation is not done but can be easily configured.
- Implemented to showcase **plug n play**

Similarly, other features can be plugged in by adding a job with the new trigger.