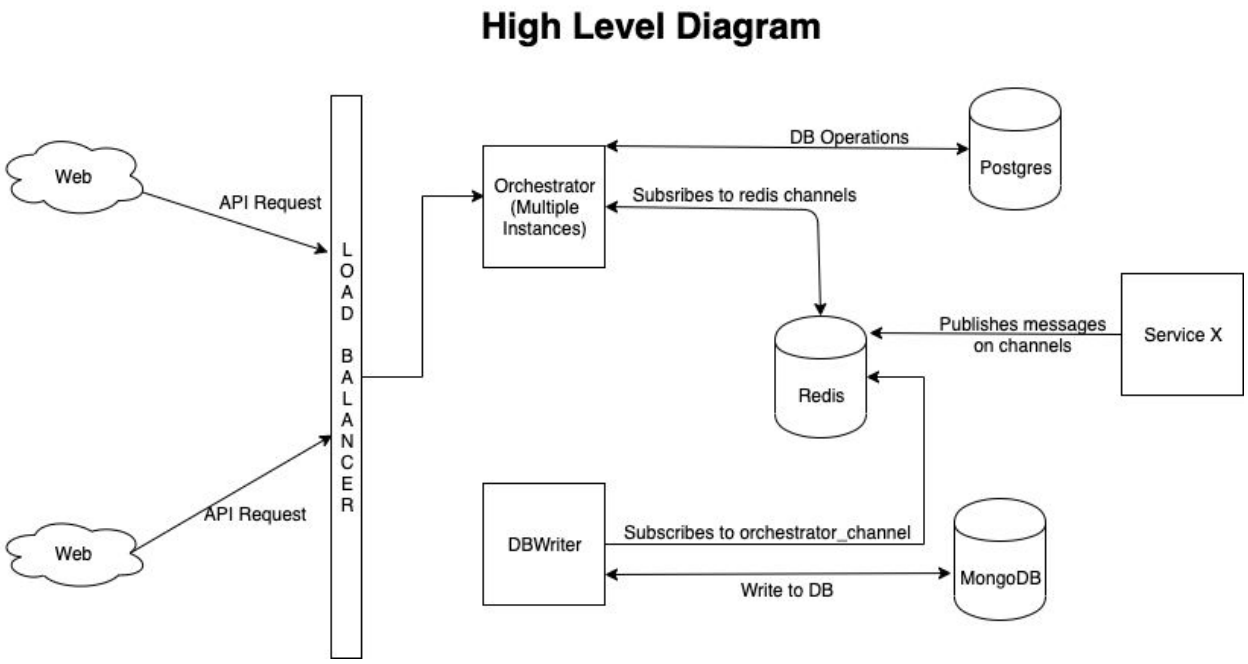
**ASSIGNMENT SOLUTION**




**Orchestrator:**

- Database: Postgres (To store the channels data)
    - Schema:

```
Table channels as C {
    id int [pk, increment] // id of the channel
    channel varchar unique // channel name
    subscription_status varchar // ENUM ["ACTIVE", "INACTIVE"]
    created_at datetime // record insertion time
    updated_at datetime // record updation time
}
```

- On server start/restart, it creates a redis connection as well as a connection to postgres. Get all the subscribed channels from the database and subscribe to them in redis.

- POST /api/v1/start-listening
    - First check if the channel is subscribed or not. If already subscribed, throw 422 or 409. Else insert an entry into the channels table and subscribe to the channel in redis.

- DELETE  /api/v1/stop-listening/channel-1
    - Again check if  the :channel_id is in the database, if  not, throw error (404 Not Found). Else unsubscribe the channel from redis and delete from the channels table in the database.

- PUBLISH message m on channel c.
    - There can be two approaches that we can follow:

**Approach 1:**

- We will spawn a child process for every subscription request, and the job of this child will be to publish the messages on the orchestrator_channel. Also we will update the record against the corresponding channel by making it's subscription_status to ACTIVE.
- The child will be killed on the DELETE /api/v1/stop-listening/:channel request.
- When this service is restarted, it will fetch all the channels from the database and resubscribe to all the channels by spawning the child process. And in this way we can handle restart process.
- Although, there would be one disadvantage in this, as we get more number of subscription requests, there will be many child processes running which we need to manage.

**Approach 2:**

- We can also run a cron-job every minute to get all the channels from the database which are inactive (subscription_status == INACTIVE) and subscribe to them in redis.
- We can use crontab at the os level as well for this.
- Even if any of the process gets killed due to any reason, it can get picked up in the next cycle of the cron-job. This would ensure fault tolerance in the system.
- The job of this process will be to read the messages on the given channel and publish it onto the orchestrator_channel.
- Once the channel is unsubscribed, we will delete the channel from the database, and kill the process.

**DBWriter:**

- Database: MongoDB
    - Schema:
        - Id (ObjectId)
        - channel (string)
        - message (string)
        - unique(channel, message)
    - The unique constraint of {channel, message} will make sure no duplicate messages are inserted into the database.

- On startup or restart, this service makes a redis connection and subscribes to the orchestrator_channel. It will also connect to the mongoDB for writing the messages.

- The subscriber will be responsible for reading the messages on the orchestrator_channel and writing those into the database.