Job Scheduler Service

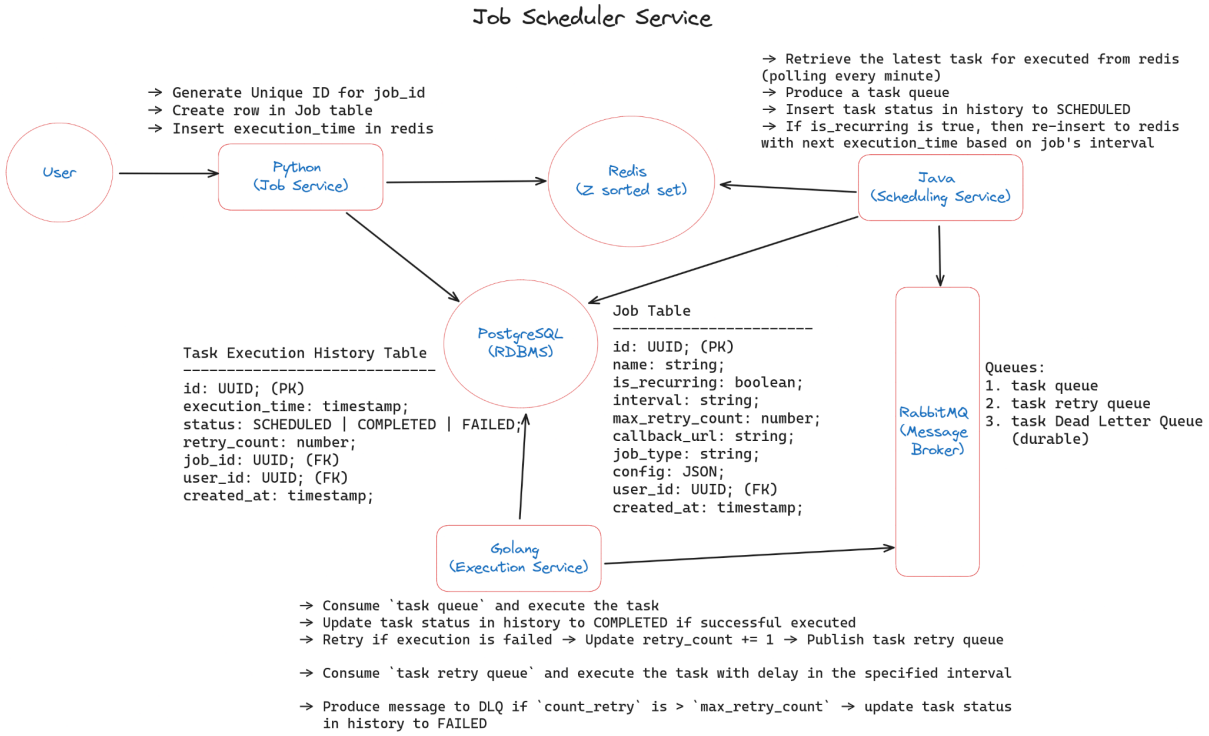
Introduction

Layanan yang dibuat pada tugas ini adalah ingin menyediakan layanan job scheduler service, yaitu service yang bertujuan untuk mengatur dan menjadwalkan *tasks* secara otomatis.

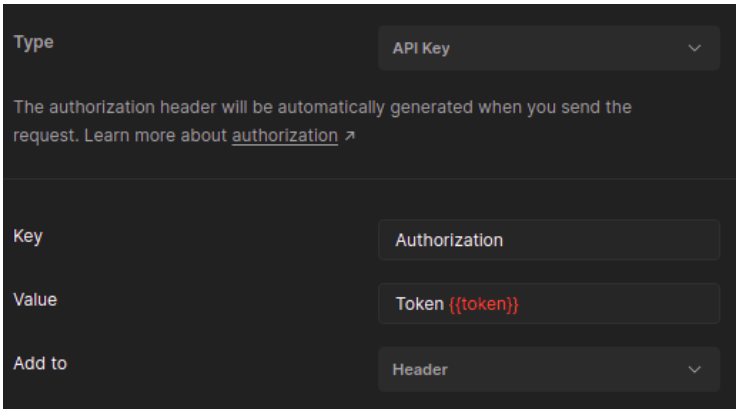
Dalam layanan job scheduler service ini, terdapat beberapa service yang dapat digunakan, seperti:

- Auth: Bertanggung jawab untuk registrasi (register) dan otentikasi (login) pengguna.
- Job API: Mengatur penjadwalan *task* yang ingin dijalankan.

Selain itu, service ini menggunakan arsitektur microservice, di mana setiap service memiliki perannya masing-masing. Setiap service dapat di*deploy* secara independen menggunakan CI/CD, memungkinkan penggunaan beragam bahasa pemrograman (*polyglot*), dan dapat diskalakan (scaled) secara independen untuk memenuhi kebutuhan memori/proses yang lebih besar.

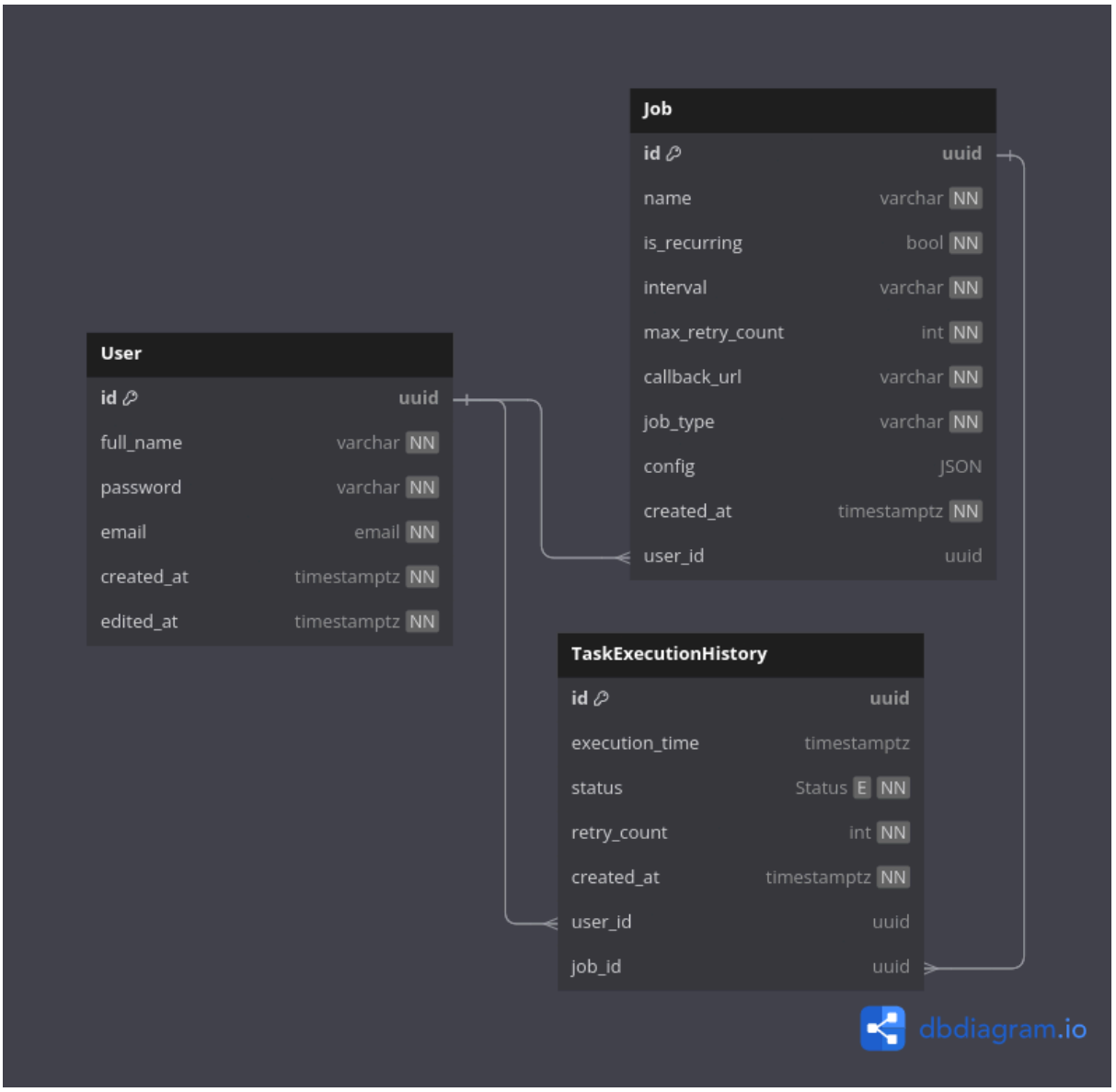


Untuk membuat job scheduler, User harus melakukan register/login terlebih dahulu untuk menggunakan service di atas. Kemudian, user dapat mengakses resource APIs pada **job_service** di atas, memerlukan *Authorization header* berupa **token-based**.



Database Design

Tahapan pertama adalah bagaimana caranya mendesain database yang tepat untuk service ini. Data yang disimpan adalah data User, Job, dan History Job.



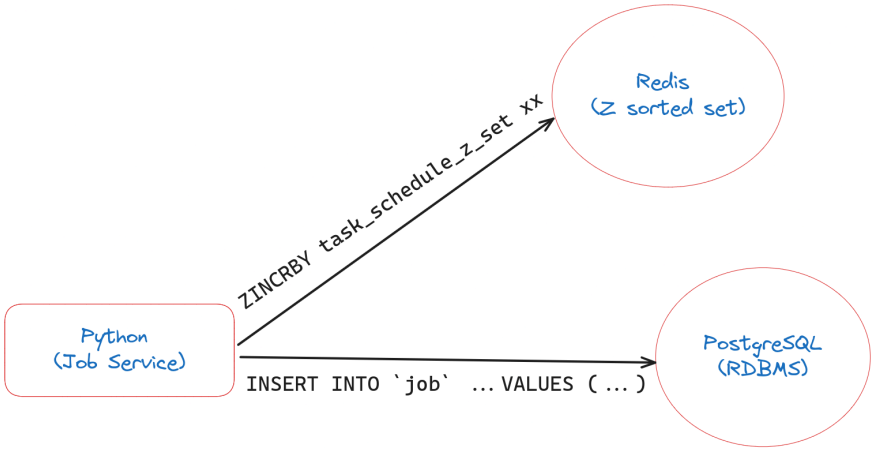
Untuk melakukan polling pada task yang perlu diambil semua tugas terjadwal yang dijadwalkan untuk dijalankan sekarang, menggunakan SQL akan menjadi *bottleneck* dan sehingga menjadi tidak *scalable.* Kita perlu melakukan query di bawah ini setiap menit:

```***query-sql***
select job_id
from job
order by job.created_at desc
limit 1;
```

Kompleksitas waktu menggunakan *approach* ini adalah **O(N log(N) * M)**, dimana N adalah banyak data di tabel job dan M adalah banyaknya job yang perlu dijalankan sekarang.

Database yang tepat untuk melakukan polling yang *scalable* juga adalah redis *sorted set*. Dengan menggunakan *sorted set,* maka kita dapat mengambil *task* yang telah dijadwalkan untuk dijalankan sekarang dengan cepat. Kompleksitas waktu *approach* ini adalah **O(log(N) * M)**, dimana N adalah banyaknya element di dalam *sorted set* dan M adalah banyaknya job yang perlu dijalankan sekarang.

Job Service (Django)



Job service adalah service utama yang akan langsung berinteraksi dengan client. Service ini menyediakan pengelolaan terkait dengan penjadwalan *task*. Secara garis besar, job service akan:

- Mengelola *task* yang telah dibuat oleh user, yaitu dapat melakukan operasi *read/write* ke dalam database.
- Saat *create job,* maka service akan generate **UUID** untuk id job dan akan menyimpan job ke dalam database postgres ke tabel **job**.
- Hitung **next_execution_time** = (current_date_time() + durasi interval) dengan timestamp menit dalam UNIX.
- Menyimpan **next_execution_time** ke dalam redis *Z sorted set*.

Jika diterjemahkan ke dalam redis-cli, maka akan seperti berikut:

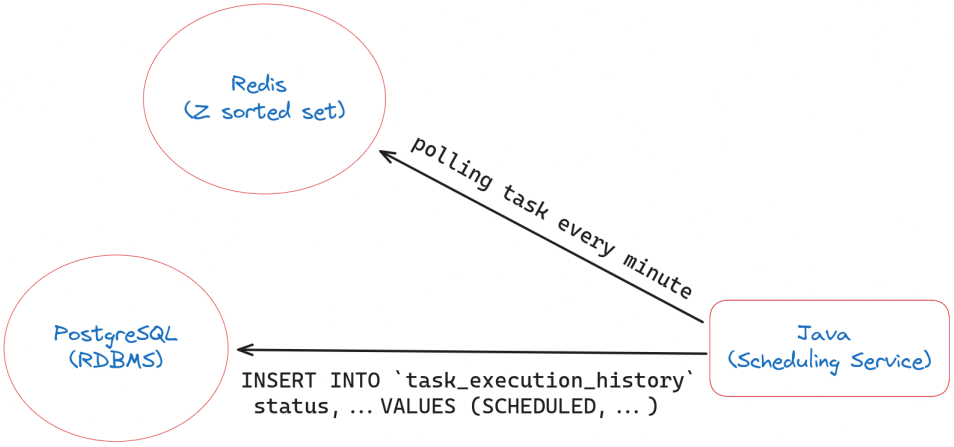
```***redis-cli***
127.0.0.1:6379> keys *
1) "task_schedule_z_set"

127.0.0.1:6379> ZADD task_schedule_z_set <next_execution_time> <job_id>

127.0.0.1:6379> ZRANGE task_schedule_z_set 0 -1 WITHSCORES
1) "c22da12f-ebc7-4255-96e0-f5a521160d90"        //RANK-1 KEY
2) "28538771"                                    //RANK-1 VALUE
...                                              //RANK-2 KEY
...                                              ...
```

Untuk KEY, merupakan UUID dari **job** yang telah dibuat. Kemudian, untuk valuenya adalah timestamp menit dalam UNIX sebagai **next_execution_time.** Dengan hal tersebut, maka penjadwalan task akan secara efisien dilakukan dengan polling.

Scheduling Service (Spring Boot)



Scheduling service adalah service yang memiliki peran melakukan polling task setiap menit yang akan dikirim ke broker untuk di*consume* oleh *execution service*. Secara garis besar, scheduling service akan:

- Melakukan polling terhadap redis *Z sorted set* setiap menit untuk suatu *task.*
- Mengirim *pending task* ke message broker dengan `task queue` untuk dieksekusi.
- Setelah *task* dijadwalkan, akan dilakukan memasukkan data pada tabel **task_execution_history** dengan status SCHEDULED.
- Service akan menghapus data job_id Jika **is_recurring** diset True oleh user, maka service akan memasukkan **next_execution_time** kembali ke dalam redis.

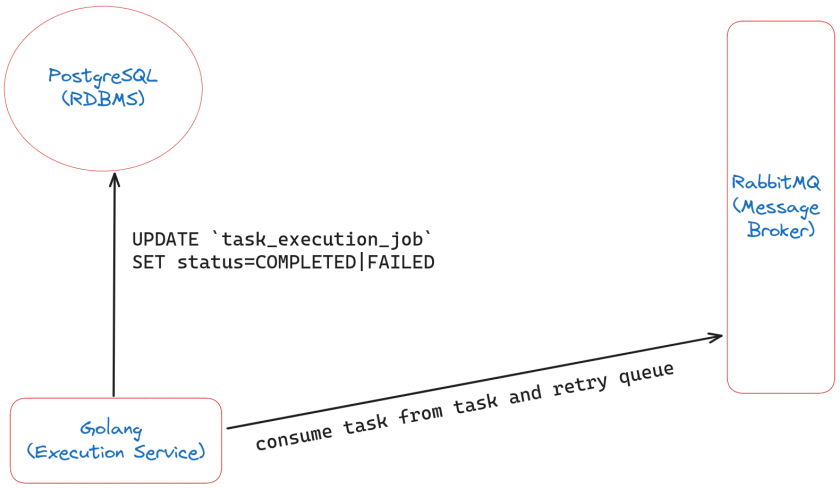
Jika diterjemahkan ke dalam redis-cli, maka akan seperti berikut:

```***redis-cli***
127.0.0.1:6379> ZMPOP 1 task_schedule_z_set MIN 1
1) "28538771"           //MIN VALUE

127.0.0.1:6379> ZADD task_schedule_z_set <next_execution_time> <job_id>
```

MIN VALUE adalah *task* yang perlu dijalankan sekarang jika value tersebut sudah kurang dari *current_date_time*(). Kompleksitas waktu ini juga sama, berjalan dalam **O(log(N) * M)**, dimana N adalah banyaknya element di dalam *sorted set* dan M adalah banyaknya job yang perlu dijalankan sekarang.

Execution Service (Golang)



Execution service adalah service yang akan meng*consume task queue* untuk melakukan sebuah eksekusi, baik *task* yang baru di*produce* atau *task* yang telah gagal dieksekusi, melalui *task retry queue.*

NB: Execution service sebenarnya dibuat oleh user yang memakai API job scheduler ini. Namun, untuk memudahkan demo, maka execution_service akan diimplementasikan di dalam sistem ini, dengan fungsionalitas mengirimkan email.

Secara garis besar, execution_service akan:

- *Consume task queue*, data yang diperoleh berupa job_id, yang nanti akan dieksekusi sesuai kebutuhan user. Dalam kasus ini, akan melakukan pengiriman email.
- *Consume task retry queue.* Jika terjadi failure ketika mengirim email, maka akan *task* akan di*produce* kembali ke message broker untuk proses *retry.*
- Jika batas *retry* sudah mencapai maksimum retry, maka *task* akan dimasukkan ke dalam DLQ (Dead Letter Queue). Kemudian, melakukan *update* pada **task_execution_history** dengan status FAILED.
- Jika *task* berhasil dieksekusi, maka akan melakukan *update* pada **task_execution_history** dengan status COMPLETED.

CI/CD

Salah satu keuntungan dari arsitektur microservices adalah dapat dilakukannya proses CI/CD secara independen sehingga dapat mempercepat proses development dan tidak saling tunggu ketika ada perubahan pada salah satu service. Proses ini secara otomatis akan publish container image ke docker.hub dan akan melakukan deployment ke instance di Google