Oban at TubiTV - Titan

by: Simon Escobar Benitez

Why Oban?

At titan we are using a handcrafted solution for processing background jobs, but that solution had a couple of problems like durability and zombie jobs.

Our current solution for background jobs has the following flow:

- From controller we enqueue some jobs to run in a different service
- Each service is polling enqueued jobs from controller and starts running them
- Once the job is finished it will trigger a sync notification to controller (via GRPC) so controller mark the job as completed/failed
- The problem now is that sometimes because timeouts, service is being re-deployed, etc, the notification isn't delivered so that job becames in a zombie job.

As a workaround for this problem we created a job process checker that is in charge of querying zombie jobs in controller (jobs that their statuses haven't changed in a period of time) and asked for their status in the corresponding service

This works but is not ideal, so I started researching about asyncrhonous jobs processing tools already implemented in elixir and found Oban

Other Options

- https://github.com/samsondav/rihanna
- https://github.com/sheharyarn/que
- https://github.com/akira/exq

So, Why Oban?

- Rihanna doesn’t support multiple queues at this moment (it will but we don’t know when)
- Exq only works with redis, and redis is not persistent (something that we need)
- Que uses mnesia and it haven’t been updated in a while (Aug 21th 2019)
- Oban has more resources than the rest of the other solutions

What Is Oban?

Oban is a robust job processing library which uses PostgreSQL for storage and coordination.

It's primary goals are reliability, consistency and observability.

It is fundamentally different from other background job processing tools because it retains job data for historic metrics and inspection. You can leave your application running indefinitely without worrying about jobs being lost or orphaned due to crashes.

Advantages Over Other Tools

Fewer Dependencies — If you are running a web app there is a very good chance that you're running on top of a RDBMS. Running your job queue within PostgreSQL minimizes system dependencies and simplifies data backups.

Transactional Control — Enqueue a job along with other database changes, ensuring that everything is committed or rolled back atomically.

Database Backups — Jobs are stored inside of your primary database, which means they are backed up together with the data that they relate to.

Advanced Features

Isolated Queues — Jobs are stored in a single table but are executed in distinct queues. Each queue runs in isolation, ensuring that a job in a single slow queue can't back up other faster queues (GenServers).

Queue Control — Queues can be started, stopped, paused, resumed and scaled independently at runtime across all running nodes (even in environments like Heroku, without distributed Erlang).

Resilient Queues — Failing queries won't crash the entire supervision tree, instead they trip a circuit breaker and will be retried again in the future.

Job Killing — Jobs can be killed in the middle of execution regardless of which node they are running on. This stops the job at once and flags it as discarded.

Triggered Execution — Database triggers ensure that jobs are dispatched as soon as they are inserted into the database.

Unique Jobs — Duplicate work can be avoided through unique job controls. Uniqueness can be enforced at the argument, queue and worker level for any period of time

- [Caveats 1.]

Scheduled Jobs — Jobs can be scheduled at any time in the future, down to the second.

Periodic (CRON) Jobs — Automatically enqueue jobs on a cron-like schedule. Duplicate jobs are never enqueued, no matter how many nodes you're running.

Job Priority — Prioritize jobs within a queue to run ahead of others.

Job Safety — When a process crashes or the BEAM is terminated executing jobs aren't lost—they are quickly recovered by other running nodes or immediately when the node is restarted.

Historic Metrics — After a job is processed the row is not deleted. Instead, the job is retained in the database to provide metrics. This allows users to inspect historic jobs and to see aggregate data at the job, queue or argument level.

Node Metrics — Every queue records metrics to the database during runtime. These are used to monitor queue health across nodes and may be used for analytics.

Queue Draining — Queue shutdown is delayed so that slow jobs can finish executing before shutdown. When shutdown starts queues are paused and stop executing new jobs. Any jobs left running after the shutdown grace period may be rescued later.

Telemetry Integration — Job life-cycle events are emitted via Telemetry integration. This enables simple logging, error reporting and health checkups without plug-ins.

Usage

[Oban Usage, Configuration and Workers]

Unique Jobs

There are a couple of ways for inserting a large number of unique Jobs, taking advantage of Oban unique options or using DB unique indexes lets discuss each approach

Oban Unique Jobs

The unique jobs feature lets you specify constraints to prevent enqueuing duplicate jobs. Uniqueness is based on a combination of `args`, `queue`, `worker`, `state` and `insertion time`. It is configured at the worker or job level.

Unique jobs are guaranteed through transactional locks and database queries: ***they do not rely on unique constraints in the database***. This makes uniqueness entirely configurable by application code, without the need for database migrations.

Option 1

We can insert each job one by one in a loop, this works well in a simple case but at titan we sometimes need to enqueue more than 300k jobs so this is not suitable as can take days

```
job = Worker.new(args, unique: [period: @one_week], queue: :myqueue)
Oban.insert(job)
```

Option 2

We can batch insert records in batches inside a loop, this works well in more complex cases but not when enqueueing more than 100k jobs as this can take hours.

- [Caveats 2.]
- [Caveats 3.]

```elixir
defmodule TestScheduler do
  @one_week 604_800

  def schedule_jobs(stream) do
    stream
    |> Stream.chunk_every(1000)
    |> Enum.each(&insert_jobs/1)
  end

  defp insert_jobs(job_ids) do
    Enum.reduce(job_ids, Ecto.Multi.new(), fn job_id, multi ->
      job =
        Worker.new(%{job_id: job_id},
          unique: [period: @one_week],
          queue: :test
        )
      Oban.insert(multi, "#{job_id}", job)
    end)
    |> Models.Repo.transaction(timeout: :infinity)
  end
end
```

Option 3

We can use Elixir concurrency and a big pool of connections to insert jobs in parallel to improve performance.

```elixir
defmodule TestScheduler do
  @one_week 604_800

  def schedule_jobs(stream) do
    stream
    |> Stream.chunk_every(1000)
    # introduce async_stream to batch insert in parallel
    |> Task.async_stream(&insert_jobs/1, timeout: :infinity)
    |> Stream.run()
  end

  # same as before
end
```

It works but still is taking a couple of hours to insert all the jobs, so, if we want to process 300k jobs as fast as possible and have unique jobs we still need to do something else

Option 4

For this option we will need to introduce unique indexes at the DB level because we are going to use `insert_all` which is a postgres feature for batch inserting records into the database, but as we need unique jobs, we need to guarantee that each inserted job is unique by the queue and its arguments.

Partial indexes

We can use partial indexes if we only need unique jobs in some queues

```
create(
  index("oban_jobs", [:queue, :args],
    unique: true,
    where: "queue = 'myqueue'"
  )
)
```

Full indexes

We can use full indexes if we need unique jobs across all queues

```
create index("oban_jobs", [:queue, :args], unique: true)
```

Using unique indexes will let us use `insert_all` and `on_conflict: :nothing` which will omit already inserted jobs

NOTE: we are not use `insert_all` manually, we can, but instead we are going to rely on Oban's implementation.

```elixir
defmodule TestScheduler do
  @one_week 604_800

  def schedule_jobs(stream) do
    stream
    |> Stream.chunk_every(1000)
    |> Task.async_stream(&insert_jobs/1, timeout: :infinity)
    |> Stream.run()
  end

  defp insert_jobs(job_ids) do
    job_ids
    |> Enum.map(fn job_id ->
      Worker.new(%{job_id: job_id},
        unique: [period: @one_week],
        queue: :test
      )
    end)
    |> Oban.insert_all()
  end
end
```

With this tunning we can insert 300k jobs in ~7sec

```
iex(18)> prev = System.monotonic_time(:millisecond)
-576460423564
iex(19)> 1..300000 |> TestScheduler.schedule_jobs()
:ok
iex(20)> next = System.monotonic_time(:millisecond)
-576460416232
iex(21)> diff = next - prev
7332
iex(22)>
```

Custom Pruner

Oban has a out of the box pruner which is in charge of deleting old jobs that are not longer needed (finished/discarded) and this can be by `maxlength` or `maxage`

- Upcoming Enhancement - Pruner Behaviour

Pruning by `maxlength` means that we will prune jobs after we reach a given amount of enqueued jobs (we are only going to prune finished/discarded jobs) this is the default setting with a value of 1000

Pruning by `maxage` means that we will prune jobs after a given period of time, old jobs that are older than the given period of time are going to be deleted

The problem now is that some jobs need to live longer than other jobs, we need to delete some scheduled jobs every day but still keep some other jobs for a couple of weeks, so we end up building our own custom pruner with the same logic as Oban's default pruner

DEMO

Questions And Answers

Given:

- we have node 1 and 2
- job A runs on node 1
- now we crash node 1 (same, without notice), and delete node 1 completely from the cluster

Question: will job A get rerun? by node 2? or will it get stuck forever? what if node 1 didn't crash but lost connection to the database (therefore no heartbeat), will the job still be processed?, will this job get processed twice?

Answer: if the node goes away or is restarted the task can be picked up by other node as the README says: **Job Safety — When a process crashes or the BEAM is terminated executing jobs aren't lost**—they are quickly recovered by other running nodes or immediately when the node is restarted. So that means jobs are going to be re-run by other node.

Oban guarantee at-least-once execution of jobs regardless of node failures, netsplits or even database restarts.

it strives to never execute a job more than once, however, this may be unavoidable in certain failure scenarios such as

- a node losing its connection to the database
- a node dying during execution of a job

For this reason jobs should be made idempotent where possible.

Question: Let's say we have multiple jobs that depends on the result of another job. Is there a way to ensure the second job to run on the same node that runs the first job? In other words, is there a way to specify the node to run on when enqueuing a job?

Answer: I don't think so, i don't know any queue or job processing system that works in this way, as they try to parallelize the jobs running on multiple processes/nodes, we would need to build a solution for this case, fork oban and try to add this feature or maybe keep using our current solution and use oban when this is not required

Question: What if we need to notify job completion (failed/succeeded)?

Answer: when we require to notify job completion we create a job in another queue (e.g notification_queue) for reporting back to the service interested in the notification, so we can retry this a high level of times. Then the other service will be consuming (polling) from that queue, jobs to mark jobs as failed/succeeded in its own service

Caveats

1. Jobs are unique in a given period of time (60s by default), that means that after that period of time, the job can be enqueued again so we need to be really careful about this and think if we need unique jobs or we can run indempotent operations for each job.
2. We can't insert infinity records to PostgreSQL at once, so we need to insert sane default number of records, in this case 1000 records.
3. As we are inserting a big amount of records to the DB we need to increase repo's timeouts as the transaction is going to take more than 15s (the default timeout)

Resources

- https://hex.pm/packages/oban
- #oban channel on elixir slack
- https://elixircasts.io/job-processing-with-oban
- https://elixirforum.com/t/oban-reliable-and-observable-job-processing/22449
- https://sorentwo.com/2019/07/18/oban-recipes-part-1-unique-jobs.html
- https://sorentwo.com/2019/07/22/oban-recipes-part-2-recursive-jobs.html

- https://sorentwo.com/2019/08/02/oban-recipes-part-3-reliable-scheduling.html
- https://sorentwo.com/2019/08/21/oban-recipes-part-4-reporting-progress.html
- https://sorentwo.com/2019/09/17/oban-recipes-part-5-batch-jobs.html
- https://sorentwo.com/2019/10/17/oban-recipes-part-6-expected-failures.html
- https://sorentwo.com/2019/11/05/oban-recipes-part-7-splitting-queues.html
- https://dev.to/calvinsadewa/implementing-message-outbox-pattern-with-oban-131