

Platformatic

View online

Download PDF

How to create a PostgreSQL based webhook system

Paolo Insogna

Node.js TSC, Principal Engineer

Don't shoot a fly with a cannon!



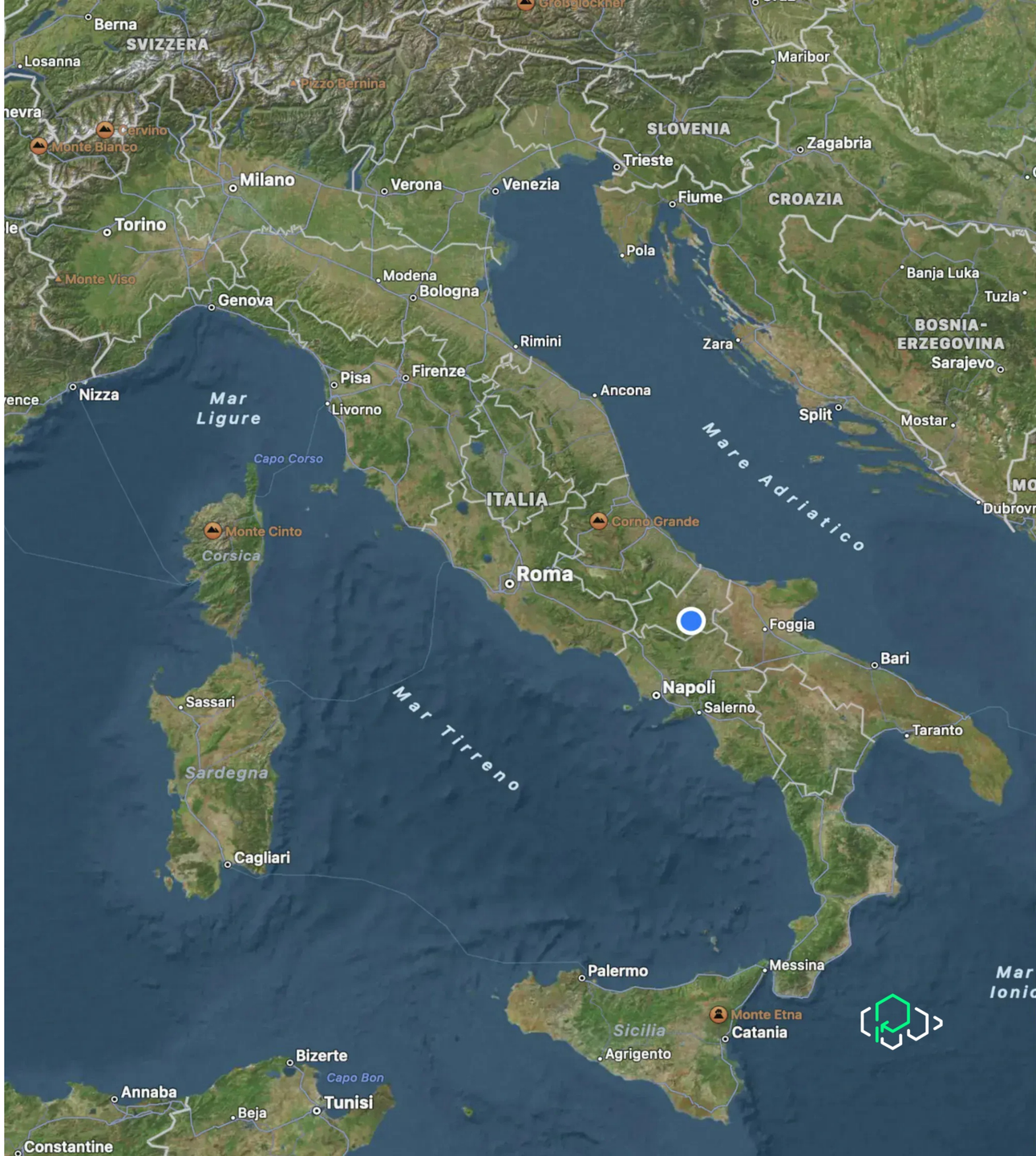
Hello, I'm **Paolo**!



Node.js Technical Steering Committee Member

Platformatic Principal Engineer

`paoloinsogna.dev` `ShogunPanda` `p_insogna` `pinsogna`



Some companies trusting us

Booking.com

Article

supabase

Case Study

SPENDESK

Case Study

Distributed data in a distributed world



We don't live a in simple world

The complexity of data cannot be handled by a single system.



Divide et impera

This represents the only possible path to have a scalable and reliable architecture.



With new powers come new challenges

How many point of failure would you like to have?

Time matters, as usual!



Distribution implies synchronization

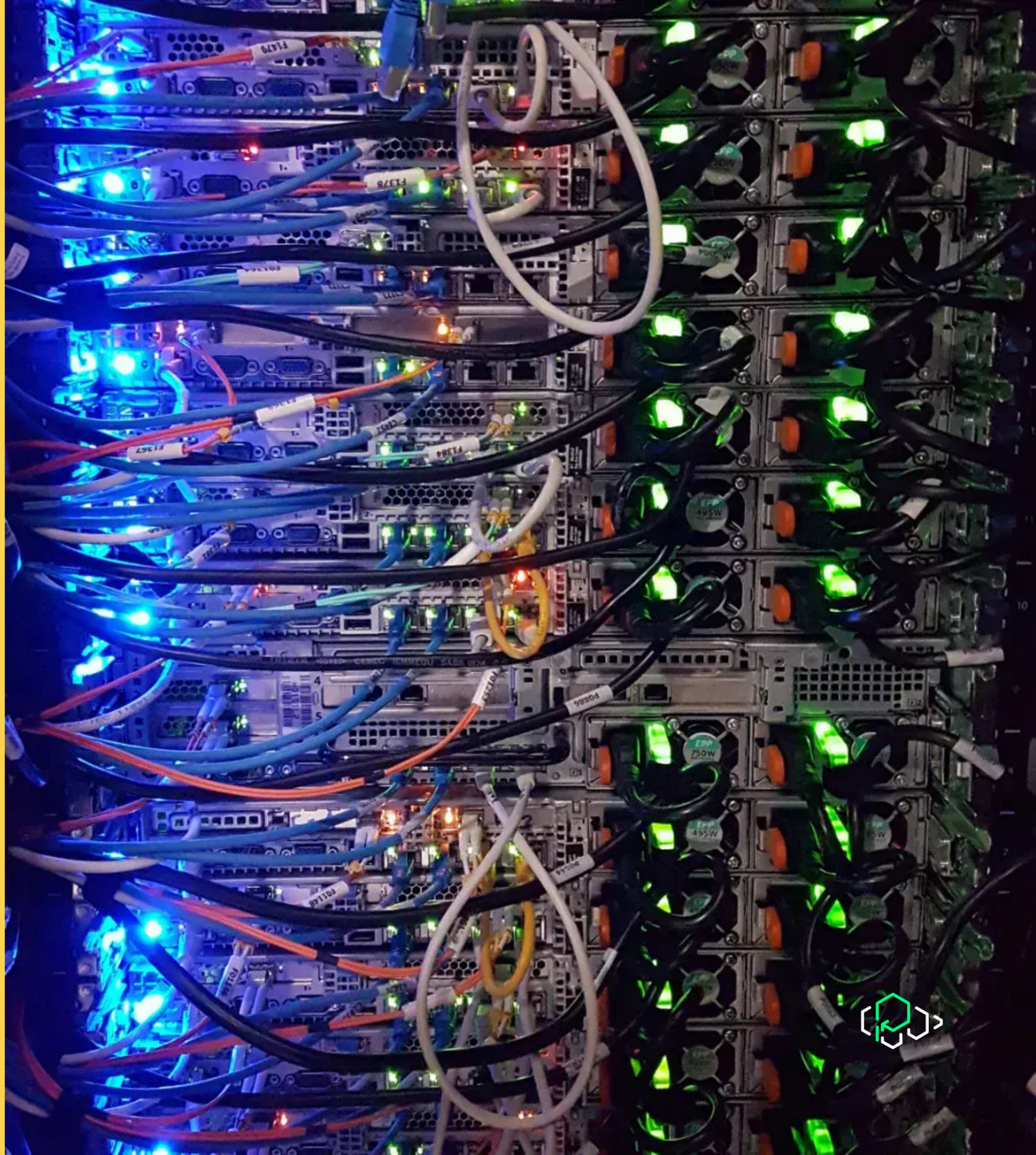
When the data is distributed, keeping a global consistent state is **hard**.



Do it right or you will fail miserably

Not all customers want to deal with things like overbooking, so be careful with your data.

How do we keep updated?



Batch updates via FTP or similar



The old good way

Still popular, especially in banks.
Your money depends on this, unfortunately.



Not realtime and fragile

Data is processed with massive delays. A failure might impact the entire batch.

Polling (pull)



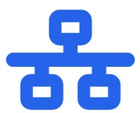
Relatively simple

Is responsibility of dependent system to periodically checks for updates.



Almost realtime

The system receives the list of updates in much shorter time.



Network intensive

A lot of uncessary requests are made, especially when updates are not frequent.

Events based (push)



Realtime

As soon as an update is available, all dependent systems are notified.



Network optimized

Not a single unnecessary byte is sent over the wire.



Delivery state is complex

The system must maintain the list of subscribers and handle delivery failures.

Which one shall we choose?



Say hello to Webhooks!



What are we talking about?

“Webhooks are user-defined HTTP callbacks. They are usually triggered by some event, such as [...] a comment being posted to a blog and many more use cases.”

Wikipedia



Let's implement a real one!



General architeture



Model as a queue

A webhook system is modeled as a queue of events.



Single target

Each queue is associated to a single URL.



Simple data structure

Few tables in a (No)SQL database are more than enough.

Yes, but which kind of queue?



Eventual

The queue might not deliver all messages.



At least once

The queue guarantees that all messages are delivered, but there might be repetitions.



Exactly once

Each message is guaranteed to be delivered only once.

Yes, but which kind of queue?

Eventual

The queue might not deliver all messages.

At least once

The queue guarantees that all messages are delivered, but there might be repetitions.

1

Exactly once

Each message is guaranteed to be delivered only once.



What about failures?



Dead letter queue (DLQ)



Limited retries

A messages is retried a limited number of times.



Separate queue for troubleshooting

Messages that cannot be delivered are moved to a separate queue.



Manual intervention

A human operator is required to evaluate the DLQ.

Cron jobs



Messages can be repeated

A system can mark a message as "to be repeated regularly".



Easy extension of the queue system

Once a message is delivered, it is enqueued again.



Standard syntax

The repetition interval syntax is really flexible.



What about race conditions?



Being equal is hard



Atomicity is hard

On distributed system, atomic access is complex and easily leads to an inconsistent state.



Concurrency is even harder

If implemented wrong, multiple peers will easily ending up operating on the same resource.



The queue model might be violated

Without coordination, policies like exactly-one might not be guaranteed.



Leader based queue system



All concur to elect a leader

All instance of the queue system must choose a leader.



Only the leader is active

No contention for resources and massive lock access attempts.



Automatic failover

When the leader resigns or fails, a new leader is automatically selected.

A simple selection implementation



Lock based

A distributed lock is used to select the leader.



Exclusivity is the key

Only one instance can hold the lock at any given time.

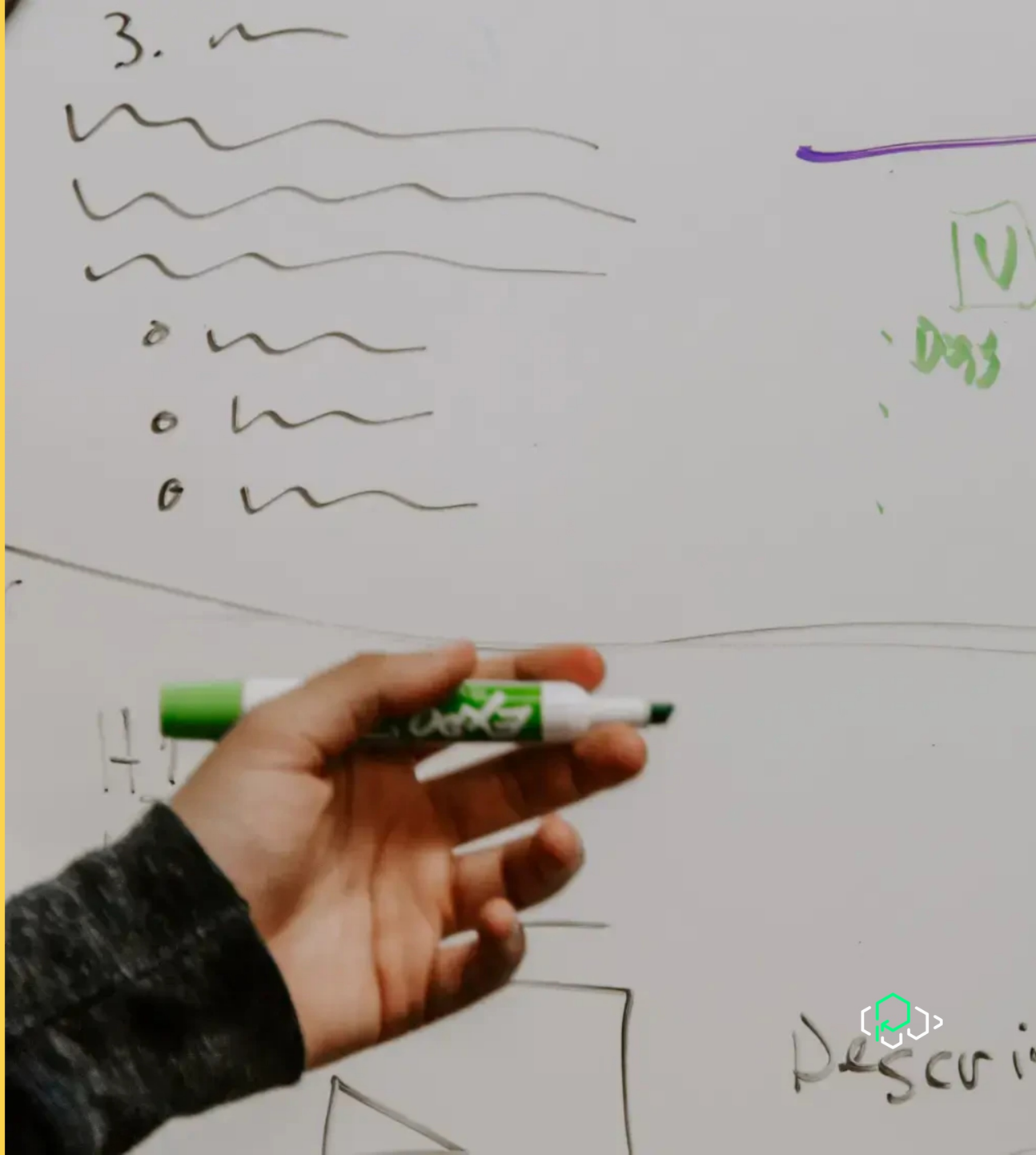


Bully-like algorithm

We are going replace the original concept of "higher ID" with locks.



How do you easily get such a lock implementation?



PostgreSQL Advisory Locks



The leader election process

1 **Lock access**
All the peers, potentially at the same time, will try to get exclusive access to the lock.

2 **Become the leader**
Only one peer receives the lock and becomes the leader, other will be denied.

3 **Leader coordinates**
The leader will start the operation, potentially delegating to other peers.

4 **Leader resigns or fail**
When the leader voluntarily resigns or fails, the other peers start a new election.



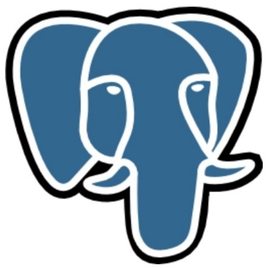
Stop talking please.
Show me the **code**!



Technical stack



Node.js



PostgreSQL



Fastify



Watt



Database schema (1/2)

```sql
CREATE TABLE IF NOT EXISTS queues (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  url VARCHAR(2048) NOT NULL,
  method VARCHAR(10) NOT NULL,
  headers JSON DEFAULT NULL,
  max_retries INTEGER NOT NULL DEFAULT 5,
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
  UNIQUE(name, url, method, max_retries)
);
```



Database schema (2/2)

```sql
CREATE TABLE pending_messages (
  id SERIAL PRIMARY KEY,
  queue_id INTEGER NOT NULL REFERENCES queues(id),
  headers JSON,
  payload BYTEA DEFAULT NULL,
  retries INTEGER DEFAULT 0,
  schedule VARCHAR(100) DEFAULT NULL,
  execute_at TIMESTAMPTZ DEFAULT NOW(),
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

CREATE TABLE completed_messages (
  id INTEGER,
  queue_id INTEGER NOT NULL REFERENCES queues(id),
  headers JSON,
  payload BYTEA DEFAULT NULL,
  retries INTEGER DEFAULT 0,
  response TEXT,
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

CREATE TABLE failed_messages (
  id INTEGER,
  queue_id INTEGER NOT NULL REFERENCES queues(id),
  headers JSON,
  payload BYTEA DEFAULT NULL,
  retries INTEGER DEFAULT 0,
  error TEXT,
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

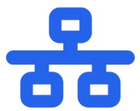


Services



Processor

The queue system that deliver messages.



API

It is used to add messages to the queue.



Composer

A `@platformatic/composer` service which orchestrates all other services.



Target

A sample Fastify application that receives the messages. It implements random failures.

Processor: Main

```
async function main () {
  const db = createConnectionPool({ bigIntMode: 'bigint' })

  globalThis.platformatic.events.on('stop', async () => {
    logger.info('Received stop event. Stopping processing jobs...')
    abortController.abort()
  })

  const abortController = new AbortController()

  while (!abortController.signal.aborted) {
    await runAsLeader(db, () => {
      logger.info('Successfully elected as leader. Starting processing jobs...')
      return processJobs(db, abortController.signal)
    })

    await scheduler.wait(1000)
  }

  await db.dispose()
}
```



Processor: Leader election and main loop

```
async function runAsLeader (db, task) {
  return db.task(async leaderConnection => {
    const [result] = await leaderConnection.query(
      sql`SELECT pg_try_advisory_lock(0) as lock`
    )

    if (!result.lock) {
      return
    }

    try {
      await task()
    } finally {
      await leaderConnection.query(
        sql`SELECT pg_advisory_unlock(0) as unlock`
      )
    }
  })
}
```



And now, let's deliver some messages!



Processor: Query to select the next message

```
export function getNextPendingMessageQuery () {
  return sql`
    SELECT
      messages.id, messages.queue_id as queue, messages.headers as message_headers,
      messages.payload, messages.retries, messages.schedule,
      queues.url, queues.method, queues.headers as queue_headers, queues.max_retries
    FROM
      pending_messages as messages
    INNER JOIN
      queues ON messages.queue_id = queues.id
    WHERE
      execute_at < NOW()
    ORDER BY
      messages.execute_at ASC
    LIMIT 1
  `
}
```



Processor: Select and deliver a message

```
async function processJobs (db, abortSignal) {
  while (!abortSignal.aborted) {
    let message
    try {
      const pending = await db.query(getNextPendingMessageQuery())
      message = pending[0]

      // Mark the message as "in-process" by clearing the execute_at field
      if (message) {
        await db.query(sql`UPDATE pending_messages SET execute_at = NULL WHERE id = ${id}`)
      }
    } catch (e) {
      /* ... */
      return
    }

    // No messages, wait for a while then continue
    if (!message) {
      await scheduler.wait(100)
      continue
    }

    // Call the target, a non 2xx will be considered a failure
    await invokeHook(db, message)
  }
}
```



Processor: Deliver a message

```
async function invokeHook (db, message) {
  try {
    const response = await fetch(message.url, {
      method: message.method,
      headers: {
        'content-type': 'application/octet-stream',
        ...message.queue_headers,
        ...message.message_headers
      },
      body: message.payload
    })

    const responsePayload = {
      status: response.status,
      headers: Object.fromEntries(response.headers.entries()),
      body: await response.text()
    }

    if (response.ok) {
      await handleMessageSuccess(db, message, responsePayload)
    } else {
      await handleMessageFailure(db, message, responsePayload)
    }
  } catch (e) {
    await handleMessageFailure(db, message, e)
  }
}
```



Why all those delays?



Notifications based scheduling



Polling is not very efficient

But it was the easiest to implement. **Forgive my laziness.**



Pushing based scheduling is way better

The processor can listen for notifications and act accordingly.



PostgreSQL pub/sub works flawlessly

Take a look at the `LISTEN` and `NOTIFY` commands. But don't forget about cron jobs.



Processor: Retry and error handling

```
function markMessageAsCompletedQuery (message, response) {
  return sql`
    INSERT INTO completed_messages (id, queue_id, headers, payload, retries, response)
    VALUES (${message.id}, ${message.queue}, ${message.message_headers}, NULL, ${message.retries}, ${JSON.stringify(response, null, 2)})
  `
}

function deletePendingMessageQuery (id) {
  return sql`DELETE FROM pending_messages WHERE id = ${id}`
}

async function handleMessageFailure (db, message, error) {
  if (message.retries < message.max_retries) {
    const timestamp = new Date(Date.now() + exponentialBackoff(message.retries + 1)).toISOString()
    try {
      await db.query(sql`UPDATE pending_messages  SET retries = retries + 1, execute_at = ${timestamp} WHERE id = ${message.id}`)
    } catch (e) {
      /* ... */
    }
  } else {
    try {
      await db.tx(async db => {
        await db.query(markMessageAsFailedQuery(message, error))
        await db.query(deletePendingMessageQuery(id))
      })
    } catch (e) {
      /* ... */
    }
  }
}
```



Processor: Reschedule a successful cron job

```
function markMessageAsCompletedQuery (message, response) {
  const serializedResponse = JSON.stringify(response, null, 2)

  return sql`
    INSERT INTO completed_messages (id, queue_id, headers, payload, retries, response)
    VALUES (${message.id}, ${message.queue}, ${message.message_headers}, NULL, ${message.retries}, ${serializedResponse})
  `
}

function rescheduleMessageQuery (message) {
  return sql`
    UPDATE pending_messages
    SET retries=0, execute_at = ${parser.parseExpression(message.schedule).next()} WHERE id=${message.id}
  `
}

async function handleMessageSuccess (db, message, response) {
  try {
    await db.tx(async db => {
      await db.query(markMessageAsCompletedQuery(message, response))
      await db.query(message.schedule ? rescheduleMessageQuery(message) : deletePendingMessageQuery(message.id))
    })
  } catch (e) {
    /* ... */
  }
}
```



Remember,
successful message
cannot be retried!



API: Message creation endpoint

```
server.route({
  method: 'POST',
  url: '/messages',
  async handler (request, reply) {
    let { queue, headers, payload, schedule } = body

    if (typeof queue === 'string') {
      /* ... Find the queue by name */
    }

    if (!Buffer.isBuffer(payload)) {
      payload = typeof payload !== 'string' ? JSON.stringify(payload) : payload.toString()
    }

    const [row] = await db.query(sql`
      INSERT INTO pending_messages (queue_id, headers, payload, schedule)
      VALUES (${queue}, ${headers ?? null}, ${payload}, ${schedule})
      RETURNING id;
    `)

    reply.code(201)
    return row
  },
  schema: {/* ... */}
})
```



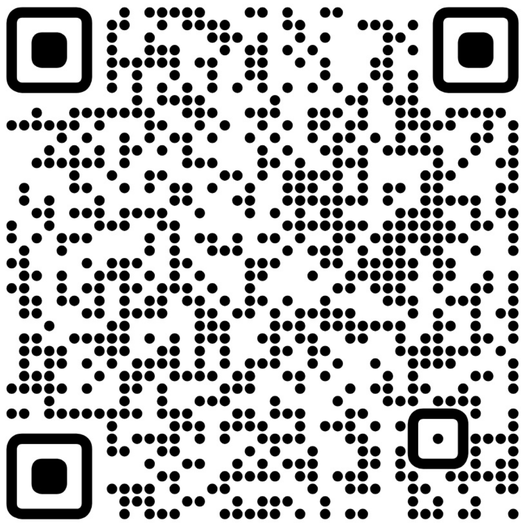
Mission completed!





Check it out!

All the code is on GitHub!



https://github.com/ShogunPanda/postgresql-webhooks

One last thing™

“Progress is man's ability to complicate simplicity.”

Thor Heyerdahl





Thank you!

Paolo Insogna

Node.js TSC, Principal Engineer

@p_insogna

paolo.insogna@platformatic.dev

Platformatic