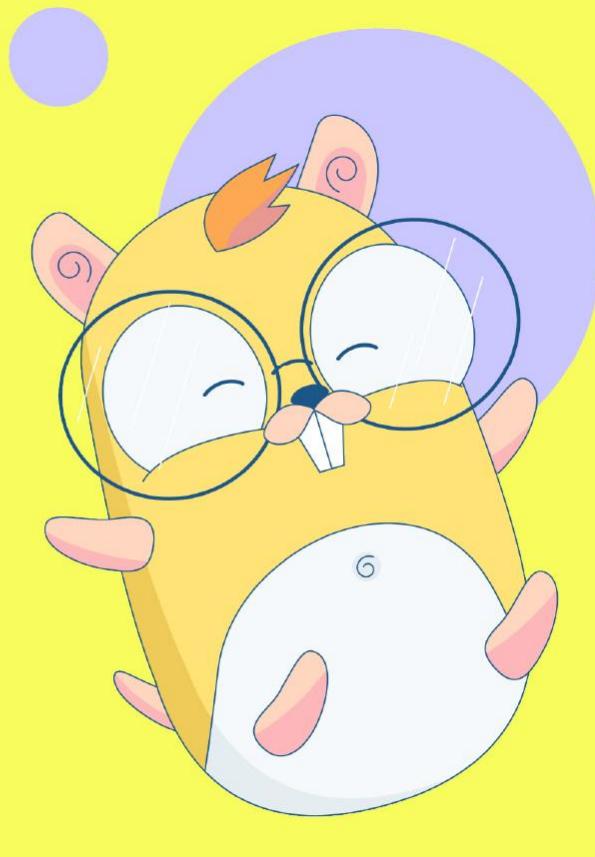


Sedlacek Tomas

CTO at **Dataddo**

pgq

postgre queues

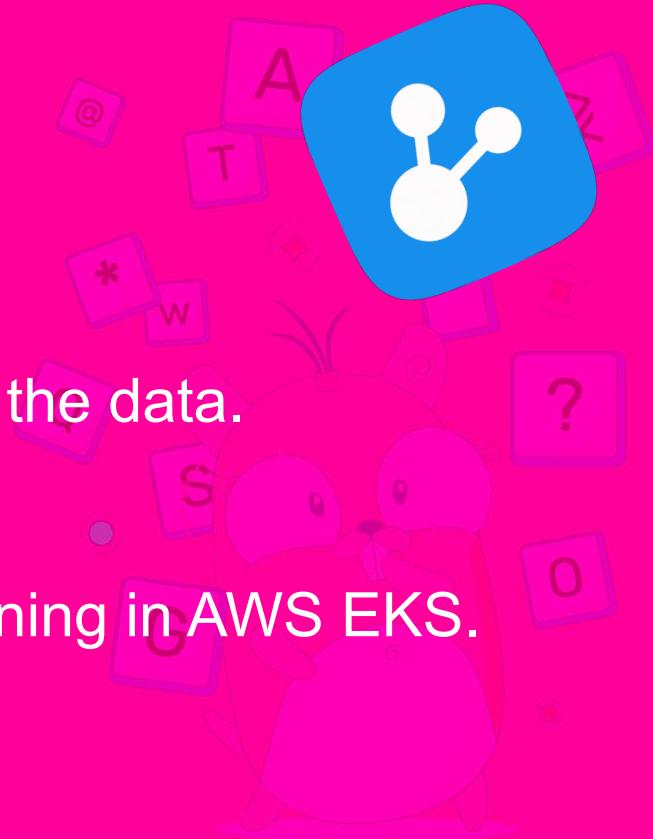


► **Dataddo.com**

A data integration platform.

Extracting, Transforming and Loading the data.
databases, http, file storages, webhooks

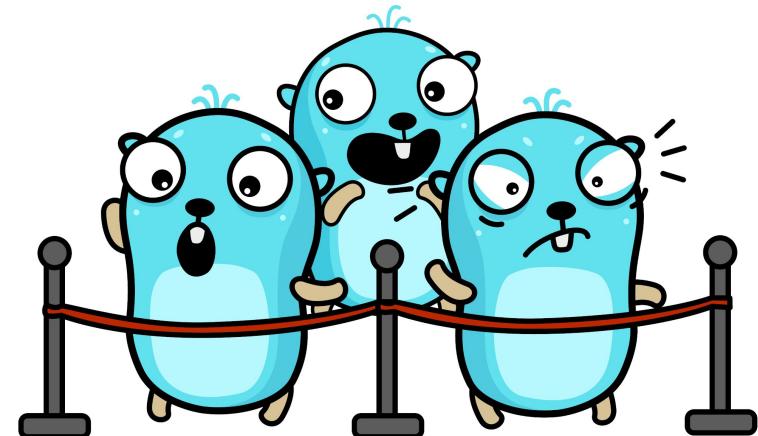
Composed of various Go services running in AWS EKS.





QUEUES

The purpose of the queues in the SW design



- **Communication** channel
- **Task scheduling**
- **Background** processing (asynchronous)
- Load **balancing** (overload prevention)
- **Throttling** and rate limiting
- Handling **peak loads**, **Scalability**, **Event-driven** architecture, Task **distribution**, **Fault** tolerance with retries

and much more...

Available Open Source

MESSAGE QUEUES [BROKERS]



- **RabbitMQ** (www.rabbitmq.com)
- **Apache Kafka** (kafka.apache.org)
- **Apache ActiveMQ** (activemq.apache.org)
- **NATS** (nats.io)
- **NSQ** (nsq.io)
- **Redis** (redis.io)
- **Amazon SQS, Google Cloud Pub/Sub, ...**

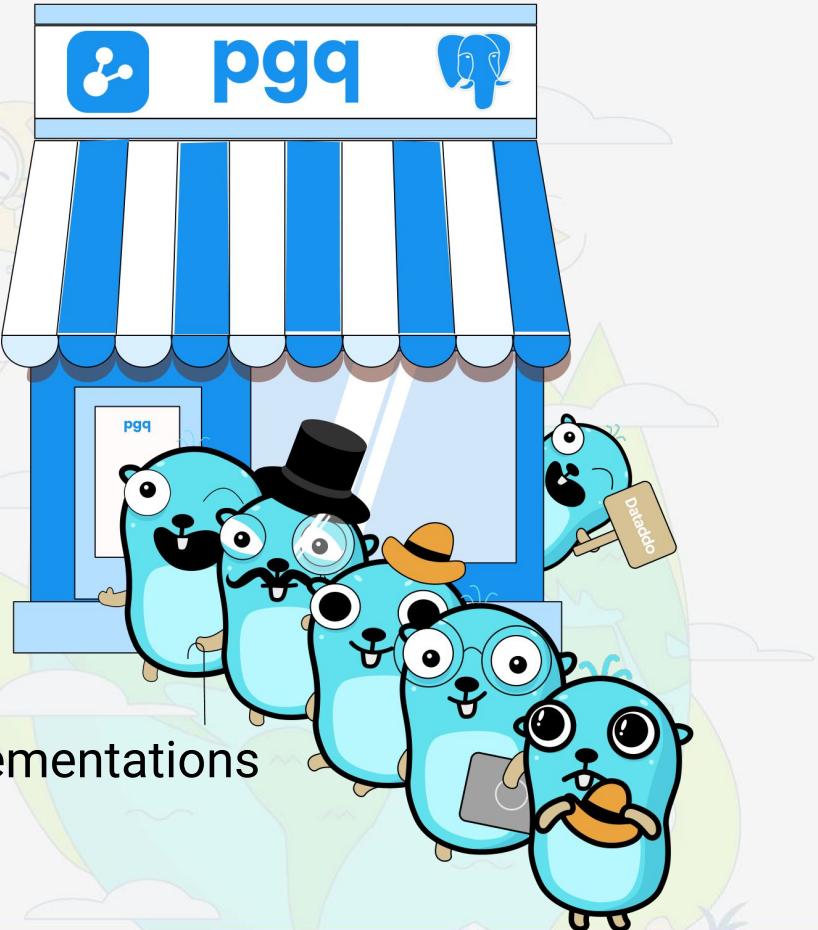
and much more...

what is the

pgq

go.dataddo.com/pgq

- Open source **go** package
- Queues on top of **postgres**
- Uses regular **SQL** statements
- Reliable and easily observable
- Basic **consumer** and **publisher** implementations

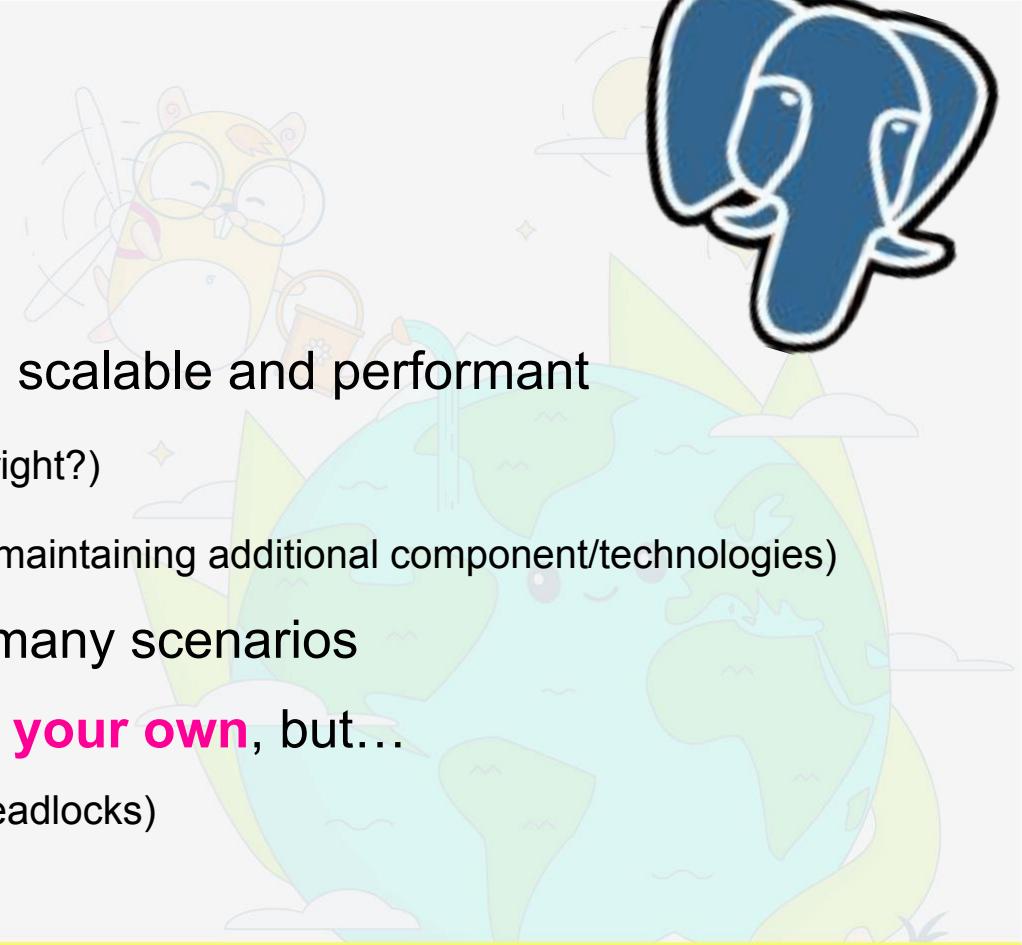


► When to pick pgq ?

- You want to build **resilient systems**
- You are already using **postgres** in your **stack**
- You do not want to **administer** another technology
- You want to easily **observe** the queues
- Your message rate is not measured in billions

▶ Why to use pgq?

- **Postgres** just works!
- **Postgres** is feature rich, scalable and performant
- **SQL** (you already know SQL, right?)
- **Simple stack** (no need for maintaining additional component/technologies)
- **Universally usable** for many scenarios
- You can implement it **on your own**, but...
(reconnects, timeouts, retries, deadlocks)



► Why to use pgq?

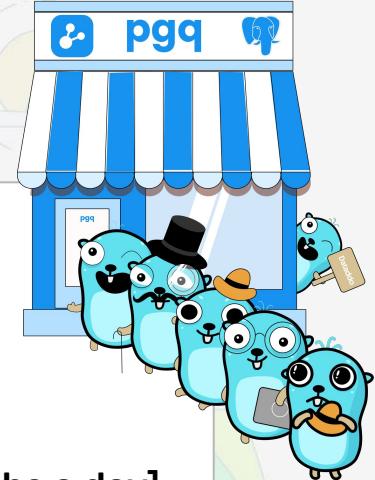
Simple stack

Keep it simple
until you can't.



► Where do we use pgq in Dataddo ?

- **Consumers of the long-running jobs**
loading & writing & processing data [200k+ of such jobs a day]
- **Consumers of the short jobs**
sending emails & saving logs & updating entities [1000k+ of such jobs a day]
- **Asynchronous apps communication**
go & php & node.js
- **Monitoring of our platform**
consumers rate & errors & peaks [AWS RDS cluster 2x db.r6g.large]

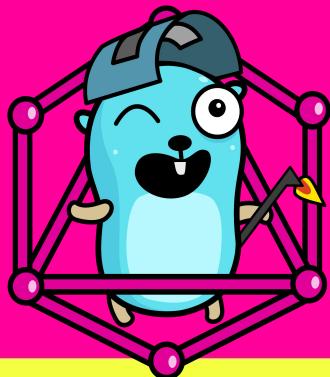


Creating the

► Queue/Table

Every **queue** is just the single postgres **table**.

Table has index for better performance.



```
CREATE TABLE IF NOT EXISTS my_queue
(
    id UUID DEFAULT,
    created_at TIMESTAMPTZ NOT NULL,
    started_at TIMESTAMPTZ NULL,
    locked_until TIMESTAMPTZ NULL,
    processed_at TIMESTAMPTZ NULL,
    consumed_count INTEGER,
    error_detail TEXT NULL,
    payload JSONB NOT NULL,
    metadata JSONB NOT NULL
);

CREATE INDEX IF NOT EXISTS my_index
ON my_queue(processed_at)
WHERE (processed_at IS NULL);
```

► The message

The message is the single row/record in the queue table.

The processed messages are kept in the queue.



```
type Message interface {  
    Metadata() map[string]string  
    Payload() json.RawMessage  
}
```

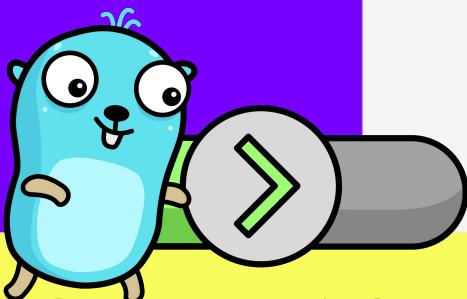
id	payload	metadata	created at	locked until	processed at	started at	error detail	consumed
UUID	JSON	JSON	Timestamp	Timestamp	Timestamp	Timestamp	String	Int
a0...	{foo:bar...}	{}	2023-...	2023-...	null	2023-...	null	1
e7...	{baz:bat}	null	2023-...	null	null	null	null	0
d2...	{go:lang...}	{o:1...}	2023-...	null	2023-...	2023-...	null	1
b6...	{lan:go...}	null	2023-...	null	2023-...	2023-...	null	1
b6...	{lag:ja...}	{f:1...}	2023-...	null	2023-...	2023-...	null	1

creating the

► Publisher

Publish message which contains metadata and payload, the consumer understands.

Note: In fact the publisher just publishes the new row to the postgres table.



```
db, _ := sql.Open("postgres", dsn)
pub := pgq.NewPublisher(db)

payload := json.RawMessage(
    `{"foo": "bar"}`)

msg := pgq.NewMessage(nil, payload)

pub.Publish(ctx, "my_queue", msg)
```

creating the

► Consumer handler

The handler is your own struct containing your custom logic needed to process the message.

```
type MessageHandler interface {  
    HandleMessage  
        (context.Context, pqq.Message)  
        (processed bool, err error)  
}
```

processed	err	description
-----	-----	-----
false	<nil>	missing failure info, but the message can be retried
false	some error	not processed, because of some error, can be retried
true	<nil>	processed, no error.
true	some error	processed, ended with error. Don't retry!

creating the

► Consumer

Consumer searches for the messages to be processed in the queue.

It updates the rows when the message is processed by handler.

```
myHandler := &myHandler{ }

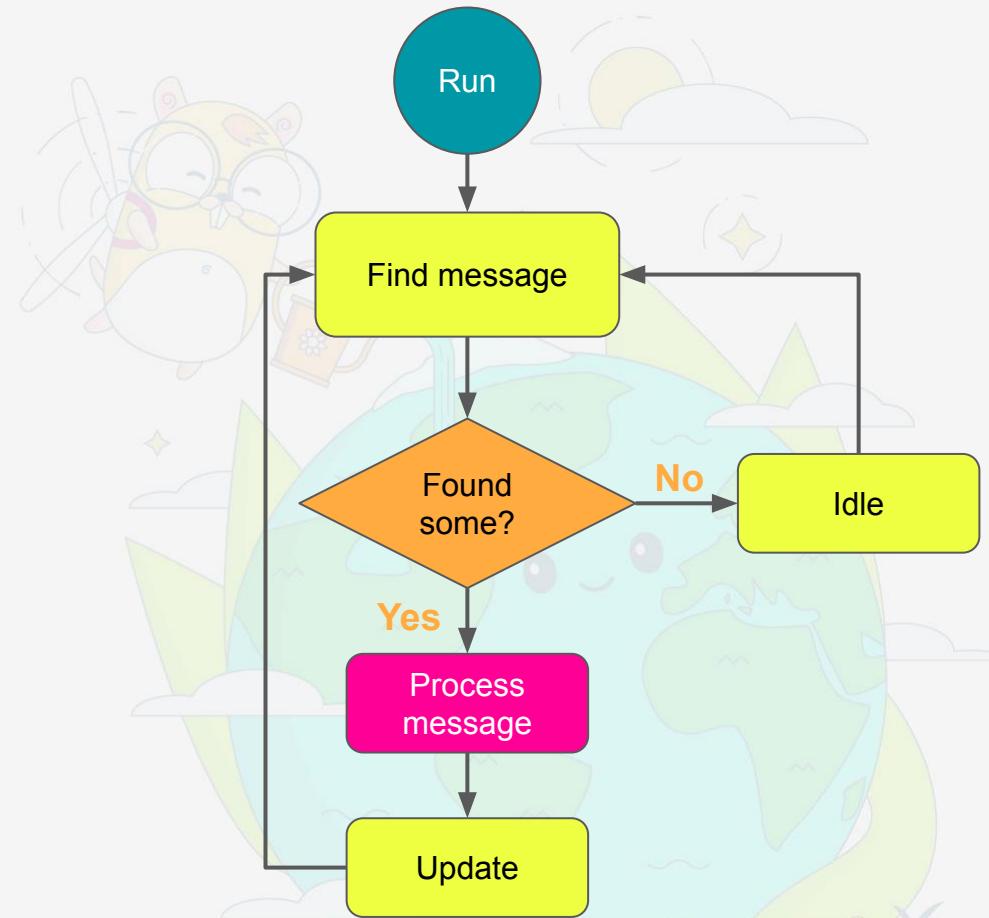
db, _ := sql.Open("postgres", dsn)
consumer := pgq.NewConsumer(
    db,
    "my_queue",
    myHandler,
)

consumer.Run(ctx)
```

Lifecycle

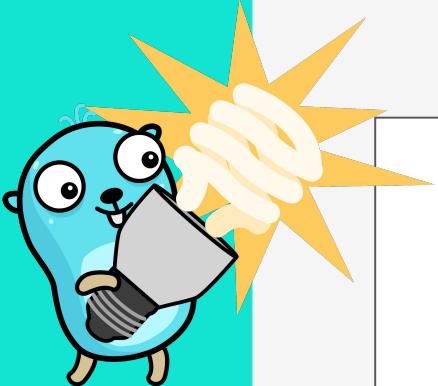
Every **consumer** polls the queue **table** in the given intervals and searches for the messages to process.

When there is no yet unprocessed message, it **idles** for a while and retries again.



how does the consumer

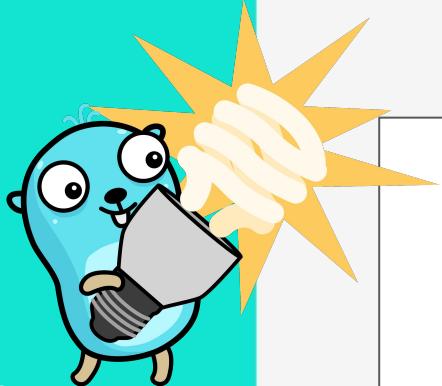
► Find message



```
UPDATE "my_queue"
  SET
    locked_until = $1,
    started_at = CURRENT_TIMESTAMP,
  WHERE id IN (
    SELECT id FROM "my_queue"
    WHERE (
      locked_until IS NULL OR
      locked_until < CURRENT_TIMESTAMP
    )
    AND processed_at IS NULL
    ORDER BY created_at ASC
    LIMIT $2
    FOR UPDATE SKIP LOCKED
  )
  RETURNING id, payload, metadata;
```

how does the consumer

► Find message



FOR UPDATE SKIP LOCKED

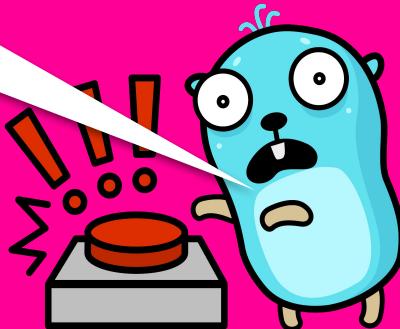
is useful in situations where multiple transactions are trying to update the same set of rows simultaneously. It locks the selected rows but skips over any rows already locked by other transactions, thereby reducing the likelihood of deadlocks.

```
UPDATE "my_queue"
  SET
    locked_until = $1,
    started_at = CURRENT_TIMESTAMP,
  WHERE id IN (
    SELECT id FROM "my_queue"
    WHERE (
      locked_until IS NULL OR
      locked_until < CURRENT_TIMESTAMP
    )
    AND processed_at IS NULL
    ORDER BY created_at ASC
    LIMIT $2
    FOR UPDATE SKIP LOCKED
  )
  RETURNING id, payload, metadata;
```

Demo time



Now we should
show them how
it works.



[alt: Gopher scared]

prerequisite

► running Postgres

You can try it on your own machines too.

Note: You need to have the running postgres db available for the demo.

*Please see the **Makefile** in the **pgq-demo** repository to get started.*

<https://github.com/dataddo/pgq-demo>

```
docker run
--name pgq-postgres
-e POSTGRES_USER=pgq
-e POSTGRES_PASSWORD=pgq
-p 5432:5432
-d
postgres:16.0
```

Recommendations

Following these principles will make your pgq usage smooth.



- Keep table **index**

Recommendations

Following these principles will make your pgq usage smooth.



- Keep table **index**
- Configure the **consumer options** according to your concrete needs
[lock duration, polling interval, max parallel]

Recommendations

Following these principles will make your pgq usage smooth.



- Keep table **index**
- Configure the **consumer options** according to your concrete needs [lock duration, polling interval, max parallel]
- Enable table **partitioning** or clear old messages when you do not need it.

Recommendations

Following these principles will make your pgq usage smooth.



- Keep table **index**
- Configure the **consumer options** according to your concrete needs [lock duration, polling interval, max parallel]
- Enable table **partitioning** or clear old messages when you do not need it.
- **Observe the queue size**, setup alerts

Recommendations

Following these principles will make your pgq usage smooth.



- Keep table **index**
- Configure the **consumer options** according to your concrete needs [lock duration, polling interval, max parallel]
- Enable table **partitioning** or clear old messages when you do not need it.
- **Observe the queue size**, setup alerts
- **Observe the errors** to detect application failures and bugs in your code

Recommendations

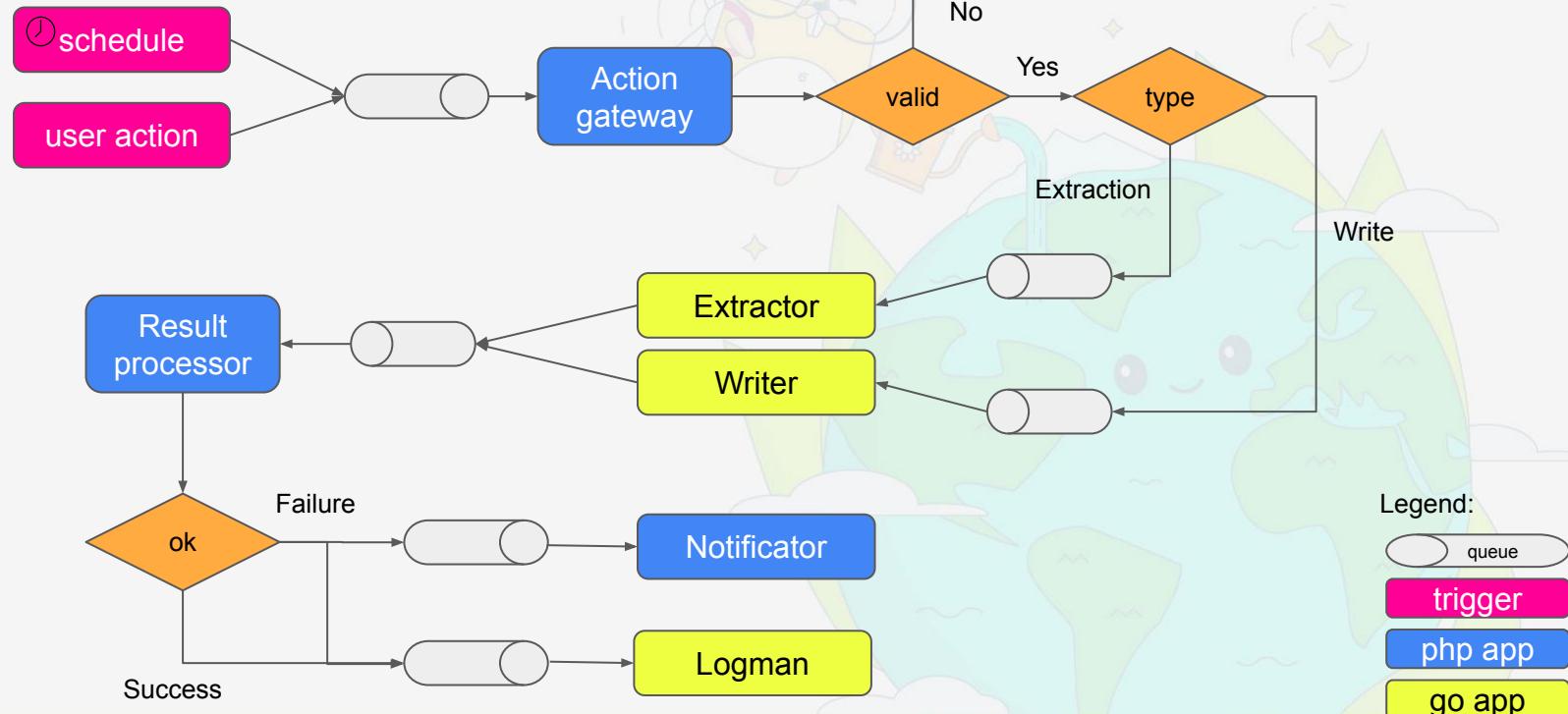
Following these principles will make your pgq usage smooth.



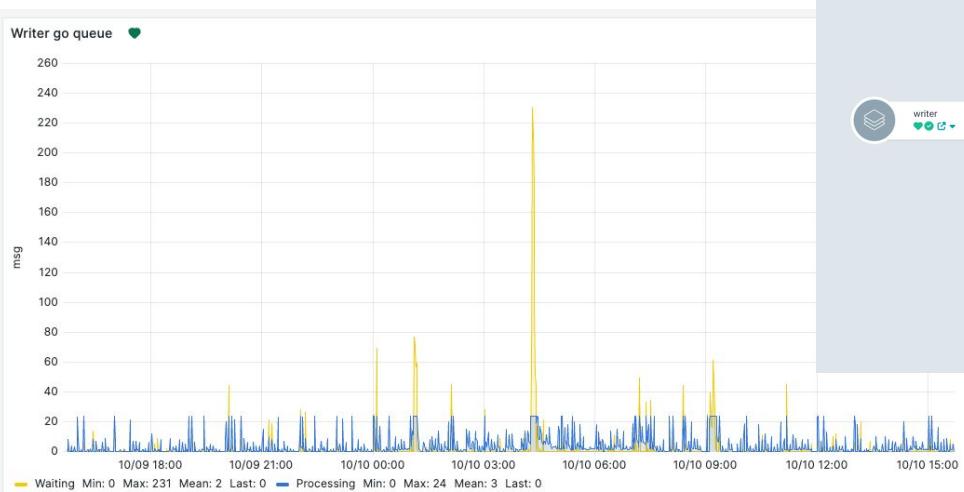
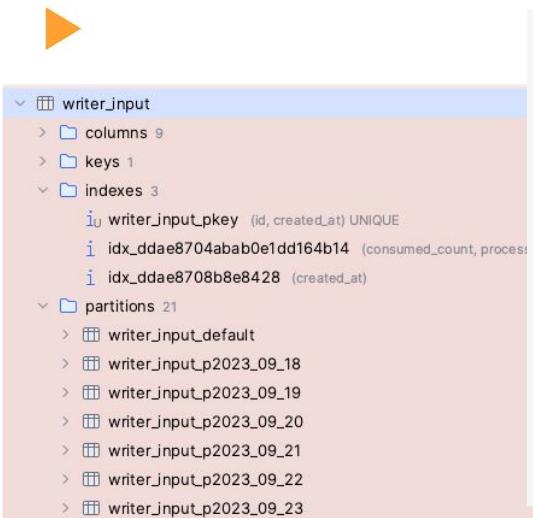
- Keep table **index**
- Configure the **consumer options** according to your concrete needs [lock duration, polling interval, max parallel]
- Enable table **partitioning** or clear old messages when you do not need it.
- **Observe the queue size**, setup alerts
- **Observe the errors** to detect application failures and bugs in your code
- Kubernetes **autoscaling** using **Keda**

The simplified dataddo

► Publishers/Consumers diagram

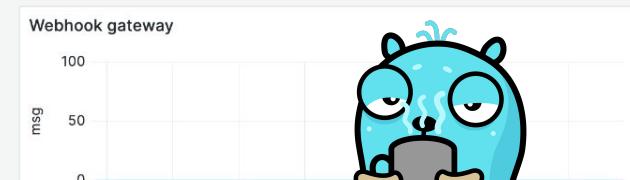
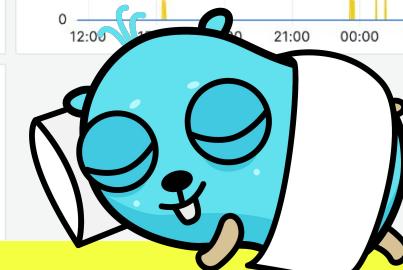
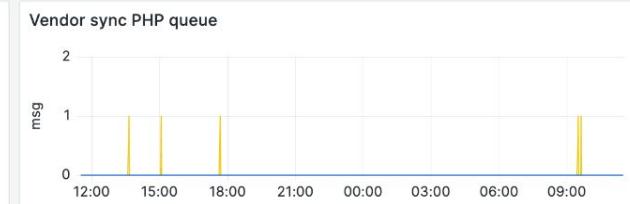
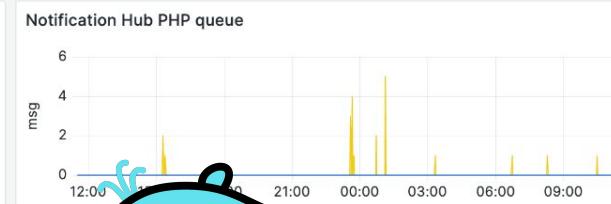
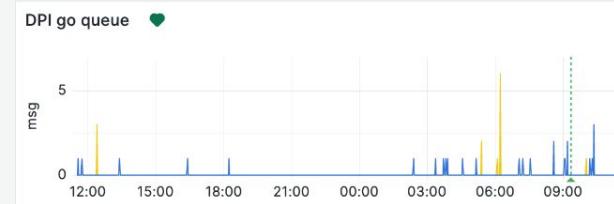
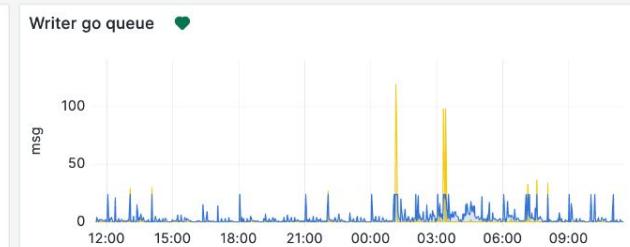
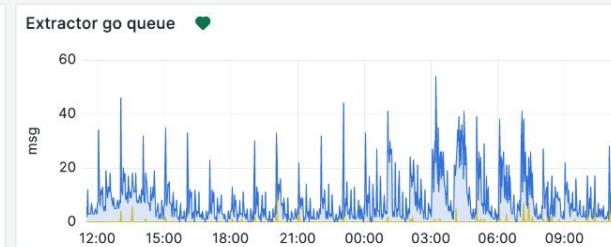
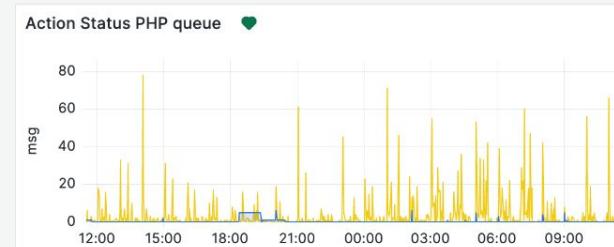
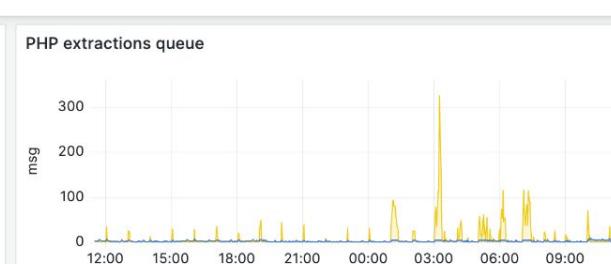
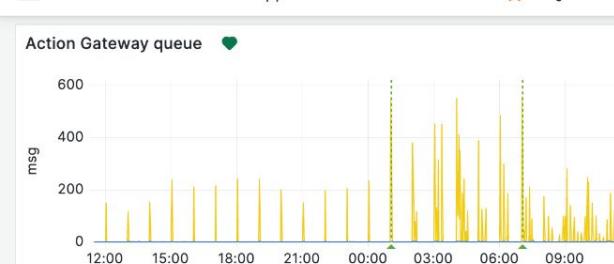


HOW DOES IT LOOK LIKE IN A REAL LIFE?



ORDER BY created_at desc											
	id	created_at	payload	metadata	locked_until	processed_at	error_detail	started_at	consumed_count		
1	bd294543-8eba-...	2023-10-10 13:40:02...	{"dpl": {"sources": [{"id": "6516c80cc09"}]}	{"app": "api-php", "actionId": "6516c85...	2023-10-10 14:40:...	<null>	<null>	2023-10-10 13:40:03 +00:00	1		
2	ee7fd03-f069-...	2023-10-10 13:35:45...	{"config": {"config": {"table": "realm...}}	{"app": "dpl", "host": "dpl-6749595495-...	<null>	2023-10-10 13:35:54 +00:00	<null>	2023-10-10 13:35:47 +00:00	1		
3	379bc0a1-69d2-...	2023-10-10 13:35:41...	{"dpl": {"sources": [{"id": "6501820609d"}]}	{"app": "api-php", "actionId": "650187b...	<null>	2023-10-10 13:36:19 +00:00	<null>	2023-10-10 13:35:42 +00:00	1		
4	2b9c5c03-255d-...	2023-10-10 13:35:40...	{"dpl": {"sources": [{"id": "64526cecbf0"}]}	{"app": "api-php", "actionId": "64526d4...	<null>	2023-10-10 13:36:55 +00:00	<null>	2023-10-10 13:35:42 +00:00	1		
5	c8cbbdca-940f-...	2023-10-10 13:35:40...	{"dpl": {"sources": [{"id": "64229fe9da0"}]}	{"app": "api-php", "actionId": "6422a17...	<null>	2023-10-10 13:35:51 +00:00	<null>	2023-10-10 13:35:42 +00:00	1		
6	19331123-f296-...	2023-10-10 13:35:40...	{"dpl": {"sources": [{"id": "6492aed5103"}]}	{"app": "api-php", "actionId": "64c773e...	<null>	2023-10-10 13:35:42 +00:00	<null>	2023-10-10 13:35:42 +00:00	1		
7	70fd7c2a-6c85-...	2023-10-10 13:35:40...	{"dpl": {"sources": [{"id": "613b2de2720"}]}	{"app": "api-php", "actionId": "613b2e0...	<null>	2023-10-10 13:35:42 +00:00	<null>	2023-10-10 13:35:42 +00:00	1		
8	48228af2-73cc-...	2023-10-10 13:35:40...	{"dpl": {"sources": [{"id": "6333fcff8b6"}]}	{"app": "api-php", "actionId": "633412b...	2023-10-10 14:35:...	<null>	<null>	2023-10-10 13:35:42 +00:00	1		
9	786f83ef-545a-...	2023-10-10 13:35:40...	{"dpl": {"sources": [{"id": "6333ff7624a"}]}	{"app": "api-php", "actionId": "6334123...	<null>	2023-10-10 13:36:17 +00:00	<null>	2023-10-10 13:35:42 +00:00	1		
10	19f92a74-cad0-...	2023-10-10 13:35:39...	{"dpl": {"sources": [{"id": "612ce3899c"}]}	{"app": "api-php", "actionId": "612ce3899c...	<null>	2023-10-10 13:35:42 +00:00	<null>	2023-10-10 13:35:42 +00:00	1		
11	c9c8cf1d-20d4-...	2023-10-10 13:35:39...	{"dpl": {"sources": [{"id": "63584feafead"}]}	{"app": "api-php", "actionId": "6358504...	<null>	2023-10-10 13:35:42 +00:00	<null>	2023-10-10 13:35:39 +00:00	1		





▶ Links and resources

PGQ:

go.dataddo.com/pgq

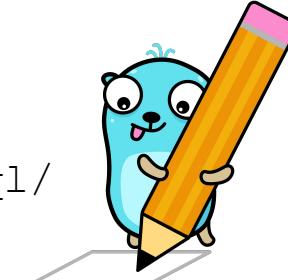
github.com/dataddo/pgq-demo

Gopher images and icons:

github.com/MariaLetta/free-gophers-pack

KEDA PostgreSQL:

<https://keda.sh/docs/2.12/scalers/postgresql/>



▶ Brain of the project

Prochac

<https://github.com/prochac>



TOMAS SEDLACEK

tomas.sedlacek@dataddo.com





Presentation slides buffer

The following slides will be used only in case of enough free time at the end of the presentation.

Inspecting the queue table

▶ **SELECT * FROM my_queue;**

id	payload	metadata	created at	locked until	processed at	started at	error detail	consumed
UUID	JSON	JSON	Timestamp	Timestamp	Timestamp	Timestamp	String	Int
a0...	{foo:bar...}	{ }	2023-...	2023-...	null	2023-...	null	1
e7...	{baz:bat}	null	2023-...	null	null	null	null	0
d2...	{go:lang...}	{o:1...}	2023-...	null	2023-...	2023-...	null	1
b6...	{lan:go...}	null	2023-...	null	2023-...	2023-...	null	1
b6...	{lag:ja...}	{f:1...}	2023-...	null	2023-...	2023-...	null	1

► The message struct

Under the hood the message contains the fields and functions necessary for operating pgq.

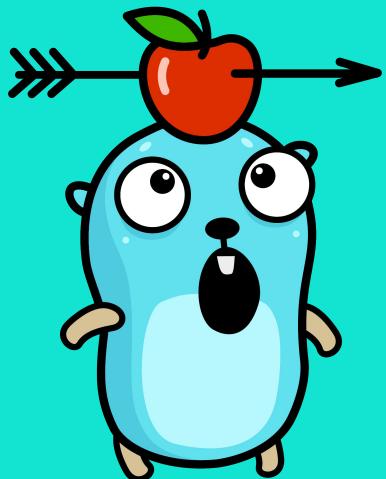


```
type message struct {  
    id uuid.UUID  
    metadata map[string][string]  
    payload json.RawMessage  
    once sync.Once  
    ackFn func(ctx Context) error  
    nackFn func(Context, string) error  
    discardFn func(Context, string) error  
}
```

pgq consumer

► Finish queries

When the message is processed (ack/nack)



```
# Ack [acknowledge] when all went fine:  
UPDATE my_queue  
  SET  
    locked_until = NULL,  
    processed_at = CURRENT_TIMESTAMP  
 WHERE id = $1;
```

```
# Nack: when something went wrong:  
UPDATE my_queue  
  SET  
    locked_until = NULL,  
    error_detail = $2  
 WHERE id = $1;
```

► Reject query

Discard the message when it is not valid.



Discard [reject] on invalid message:

```
UPDATE my_queue
  SET
    locked_until = NULL,
    processed_at = CURRENT_TIMESTAMP,
    error_detail = $2
  WHERE id = $1;
```