# Database-Backed Queues:

At Managed By Q, we try hard to stay focused on the challenges unique to our core domain. We can’t remove complexity inherent in our software systems, but we can use tools that make the complexity easier to manage.

One such tool we introduced recently is the database-backed queue — that is, using a table in a relational database as a queue for asynchronous jobs. We use DB-queues in production to ensure data integrity, prevent bugs, and make our systems easier to reason about. This post will explain why we use database-backed queues, and how we wrote a thin library to suit our purposes.

Background

Like most businesses, our technology is powered by a variety of datastores stitched together by application code. The backbone of our application is our relational databases, but we also use queues for asynchronous background jobs and inter-service messaging. For asynchronous background jobs, we use [Celery](http://www.celeryproject.org/) (a Python task-running library) with [RabbitMQ](https://www.rabbitmq.com/) as the broker. For inter-service messaging, we do pubsub with Amazon’s [SNS/SQS](https://docs.aws.amazon.com/sns/latest/dg/sns-sqs-as-subscriber.html).

Let’s look at an example

Here is what happens in one of our services when a vendor invoices a customer for a project:

1. Receive web request.
2. Mark the project as “ready for invoicing.”
3. Enqueue a job to notify the customer being invoiced.
4. Publish a message for other services to consume asynchronously.
5. Send response.

If we only think about the datastores and ignore everything else, the above example looks like this:

1. Receive request.
2. Write to PostgreSQL.
3. Write to RabbitMQ.
4. Write to SNS.
5. Send response.

In our old architecture, we would sometimes write to some datastores but not others. This could happen for a variety of reasons: network issues, AWS outages, or runtime errors — just to name a few. These issues are time-consuming to debug and manually resolve, so we started thinking about ways to prevent them altogether.

Finding the right solution

We wanted to solve this problem without using distributed transactions, which have heavy performance and availability penalties, and without having to replace our existing infrastructure. We considered either tailing the Postgres write-ahead log or using a table as a job queue. We opted for the database-backed queue due to the simplicity of the implementation, as well as interoperability between multiple relational databases (our implementation works with any Django-supported database).

With our database-backed queue, the above workflow looks like this:

1. Receive request.
2. Write to project table.
3. Write to queue table (job to forward an async task to RabbitMQ).
4. Write to queue table (job to publish a message to SNS).
5. Send response.

All 3 writes happen with a single database transaction. We rely on the atomicity guarantee of the transaction to ensure that everything happens or nothing happens. Then we rely on our queueing library to guarantee that the jobs on the queue get asynchronously replicated to RabbitMQ and SNS. Since our DB-queue guarantees at-least-once delivery, we can be confident that items in the queue will reach their downstream destinations.

Our implementation

For our database-backed queue, we wrote a thin Django package we call mbq.atomiq. The library has three main components: a queue table, a producer function, and a consumer. Our consumer is single-threaded and pulls one task off the queue at a time. We only run one consumer, which can push approximately 100 tasks/second to RabbitMQ. There’s plenty of room for optimization here, but so far we have only come near this limit during large backfills.

We intentionally avoided writing a general-purpose asynchronous task running library. Instead, we built a simple pass-through to RabbitMQ and SNS. This means that our consumer does not run any application code, and each job is either “Forward this job to RabbitMQ”, or “forward this message to SNS”. We can therefore rely on the more advanced features of Celery without having to reimplement them.

Manually resolving failed tasks

To handle manual replaying of failed tasks, mbq.atomiq comes with a Django admin view. We have separate views for RabbitMQ and SNS, so we can tailor them to display the most useful information for each type of task. The admin views display error information, including links to Rollbar for more detailed info, and provide functionality for manually replaying and deleting tasks. Our consumer tries each task three times with exponential backoff before putting them on the error queue. Note that since our mbq.atomiq consumer doesn’t execute application code, tasks end up on the failure queue very rarely.

Monitoring queue health

To monitor the health of our queues, our consumer periodically sends metrics to Datadog. We alert on wait time (the time between enqueueing a task and the task being executed for the first time), the age of the oldest task in the queue, and queue length. We also alert the owning team when one of their tasks ends up on the failure queue.

Note that although the ideas in mbq.atomiq are reusable, there are many Managed By Q-specific parts to the implementation. Check out [the repo on Github](https://github.com/managedbyq/mbq.atomiq)!

Final thoughts

Database-backed queues are how we stopped trying to solve distributed systems problems with application code. We no longer have to debug and manually resolve partial writes that make it to some datastores but not others. This frees our developers to focus on the important things — our core domain problems.