RABBITMQ IS MORE THAN A SIDEKIQ REPLACEMENT

Stanko Krtalic Rusendic

github.com/stankec
@monorkin
hey@stanko.io
stanko.io

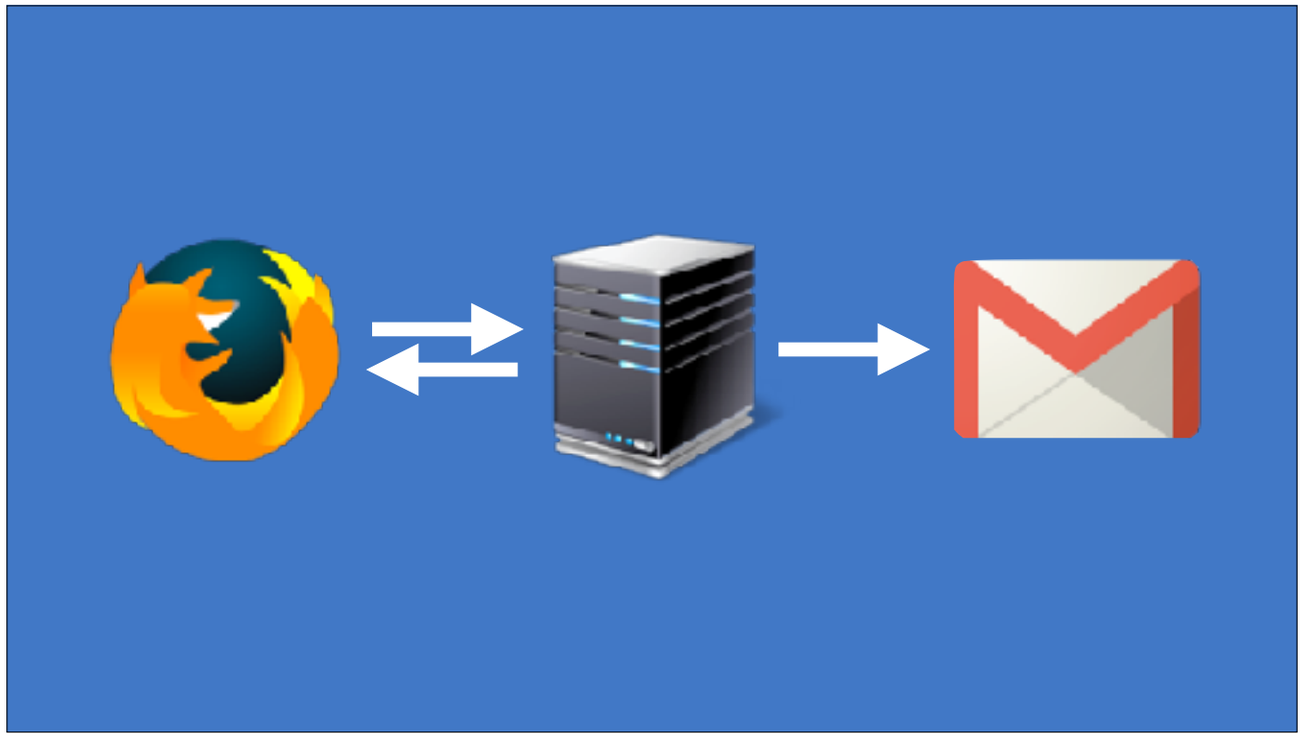
Before we continue, a quick introduction.
My name is Stanko. I am a software engineer at FloatingPoint.
I mostly work with Ruby, Elixir and Rust.

01 WORKERS

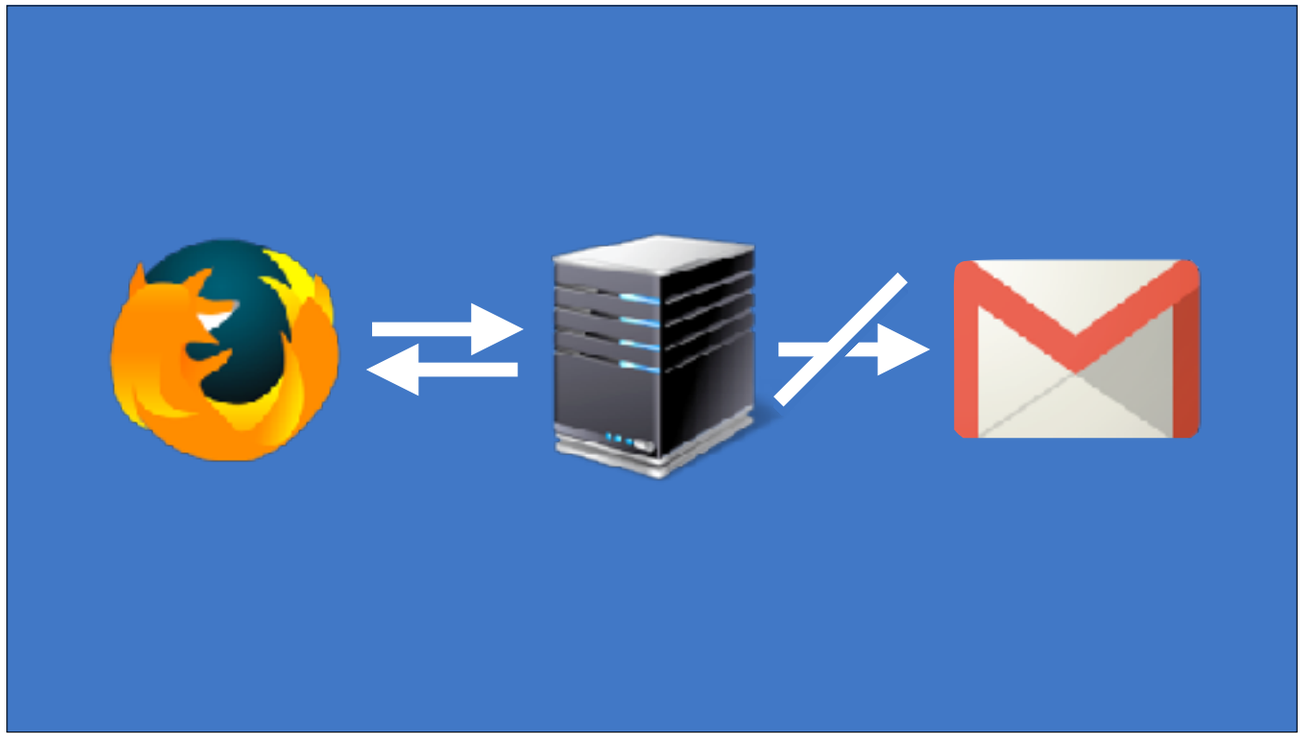
Ok, so why do we need Sidekiq or RabbitMQ? Well let’s start from what they have in common. You can use them for background workers.

Asynchronous execution

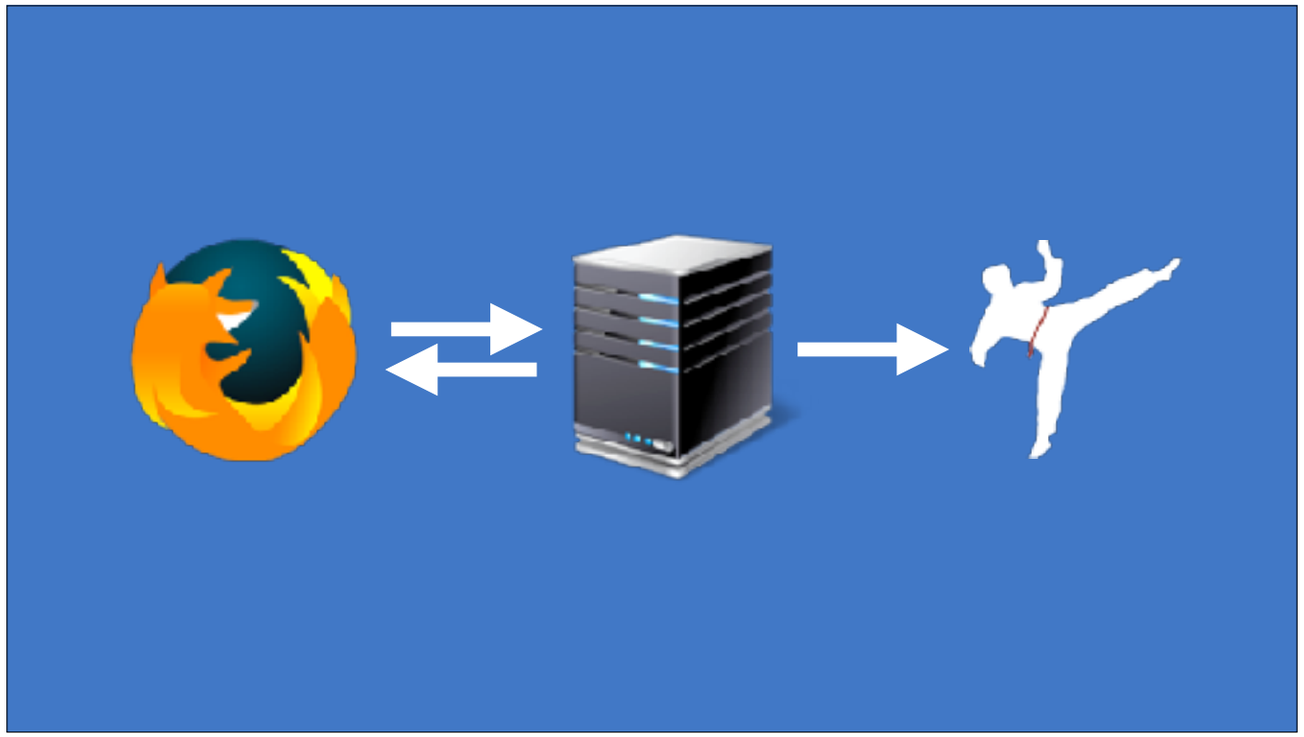
Background workers enable you to execute code asynchronously and in parallel.



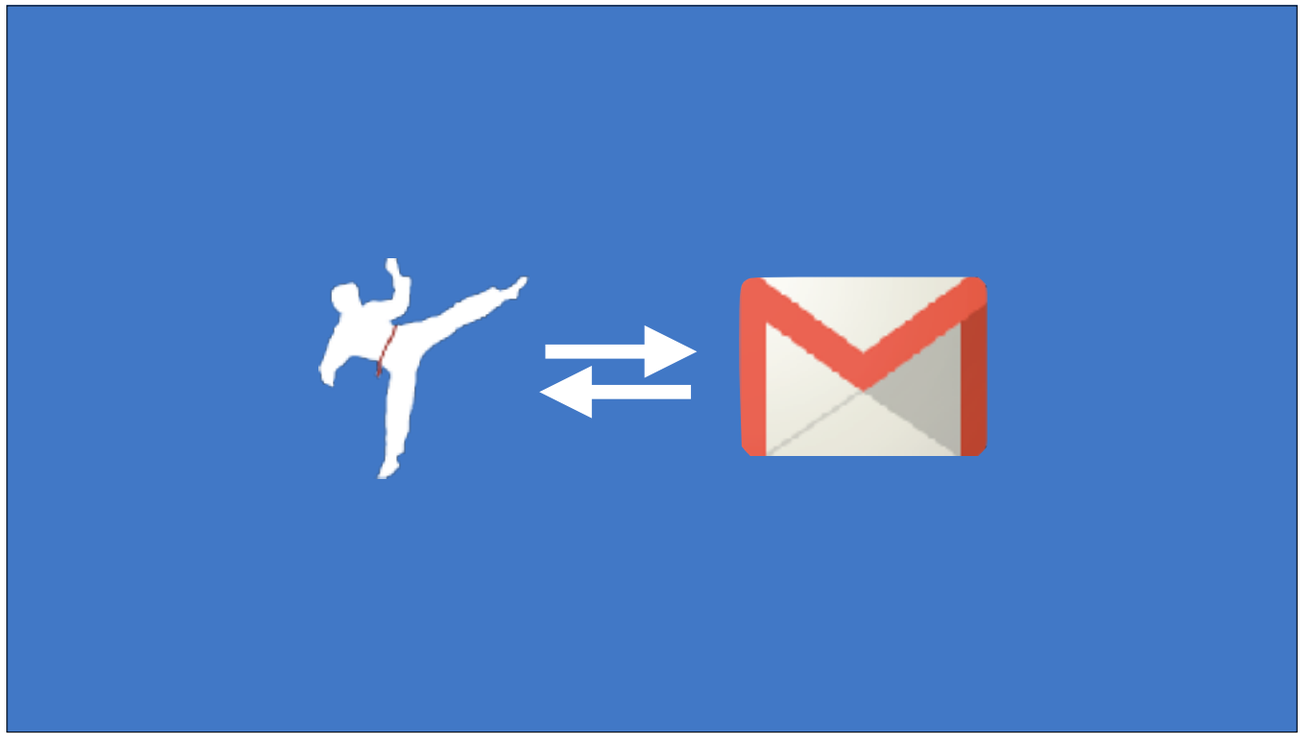
A typical example for using background workers is sending out mail, e.g. a signup confirmation.
So after you signup through the browser, the server creates a User in the database for you, and sends you an email with some instructions. We can do it the way it’s illustrated above and execute everything synchronously. But!



What if the mail fail to be sent for whatever reason? Should we fore the user to signup again? How do we cleanup our state, we have the user already in the database? Should we retry? If yes how many times? How long will the user wait?

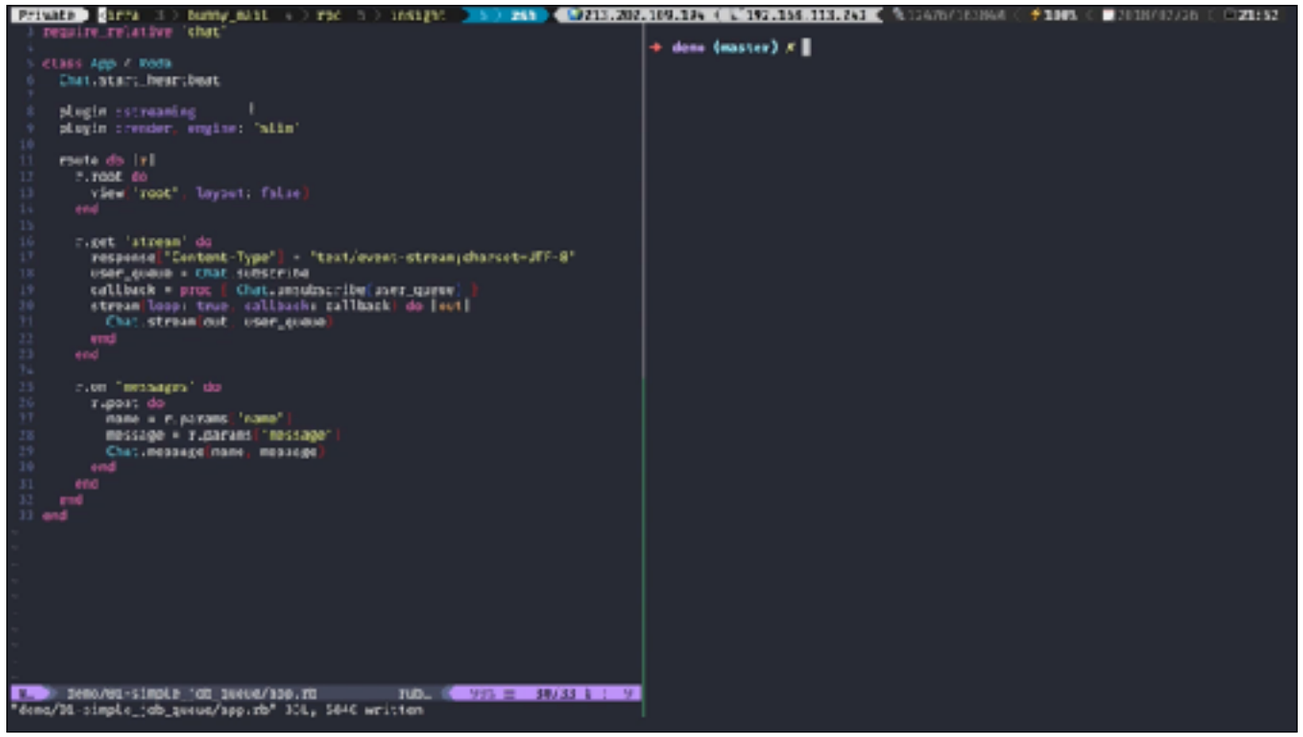


To solve those problems we utilise a background worker. Or in simple words a detached process. In the case of the previous example, instead of sending out the mail immediately, we put it in a queue of things that need to be done eventually. In this case we use Sidekiq as our background worker solution.

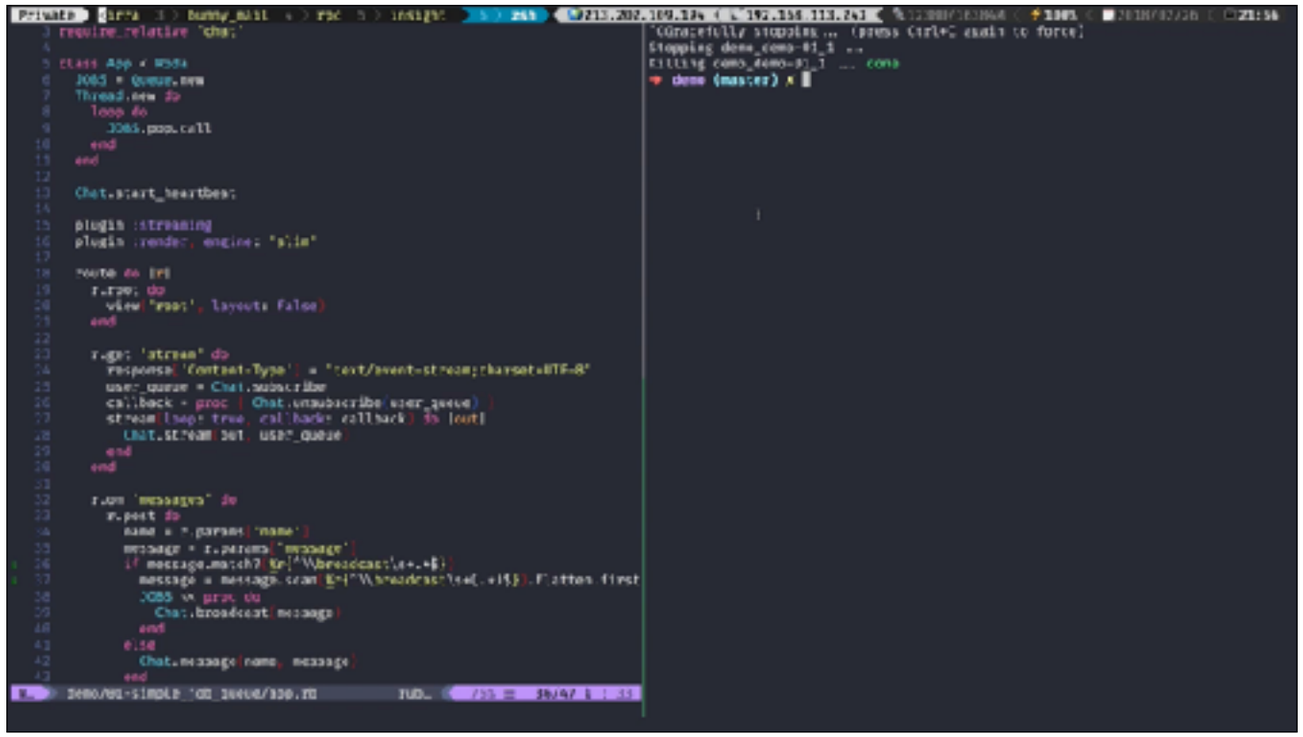


Sidekiq will at some point read the job and send out the email. If it fails it will just retry until it succeeds. This is a huge plus!

But do we really need Sidekiq to do this?



[LIVE CODING]
It’s not a lot of work to implement a simple job queue.
First we create the Queue itself, for this I will use the standard Ruby Queue class.
Then we implement a worker, in this case a thread that instantly pops the queue.
And finally we create the job.
First we check if the message is a broadcast command - commands start with a \.
Then we add a proc to the JOBS queue and that’s it!



[DEMO]
We can still send messages as normal. But now if we prefix the message with \broadcast it has special styling applied to it!

Downsides?

Ok… So why do people use Sidekiq if this is so easy to implement in plain Ruby?

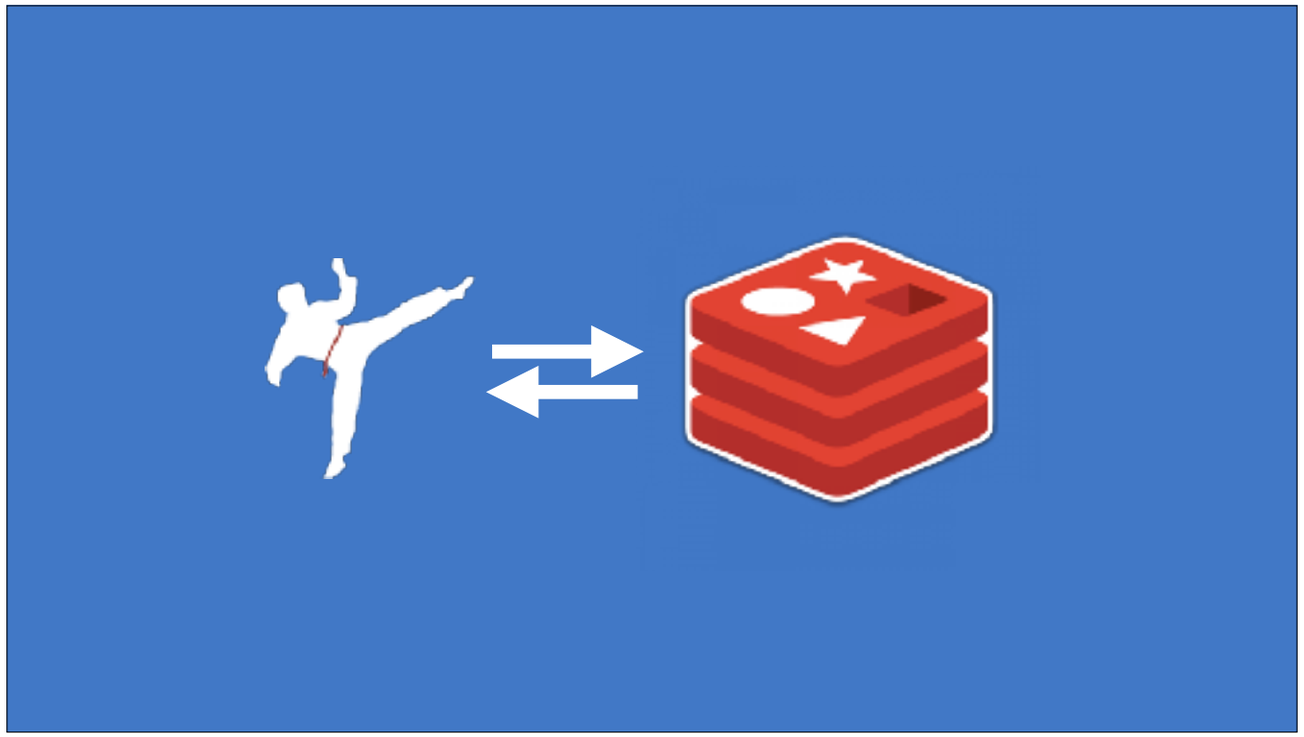
Debugging

Well, this is hard to debug. It’s not exactly trivial to inspect the contents of the queue.

Load balancing

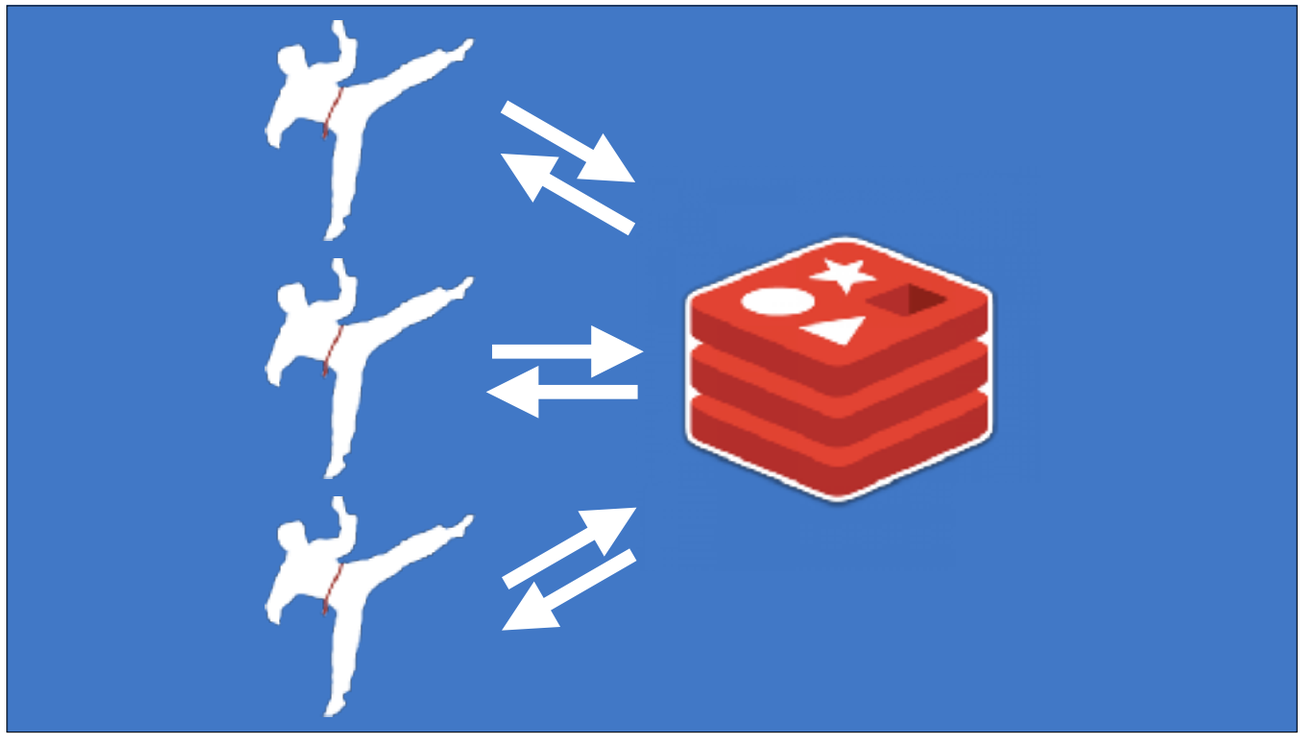
On the other hand this queue is Ruby prices specific. If I create 10 instances of this app each would have it’s own queue and it’s own workers. If one is shot down it’s jobs are lost.

Note that this isn’t impossible to implement, but it would quickly spiral into a project of it’s own.

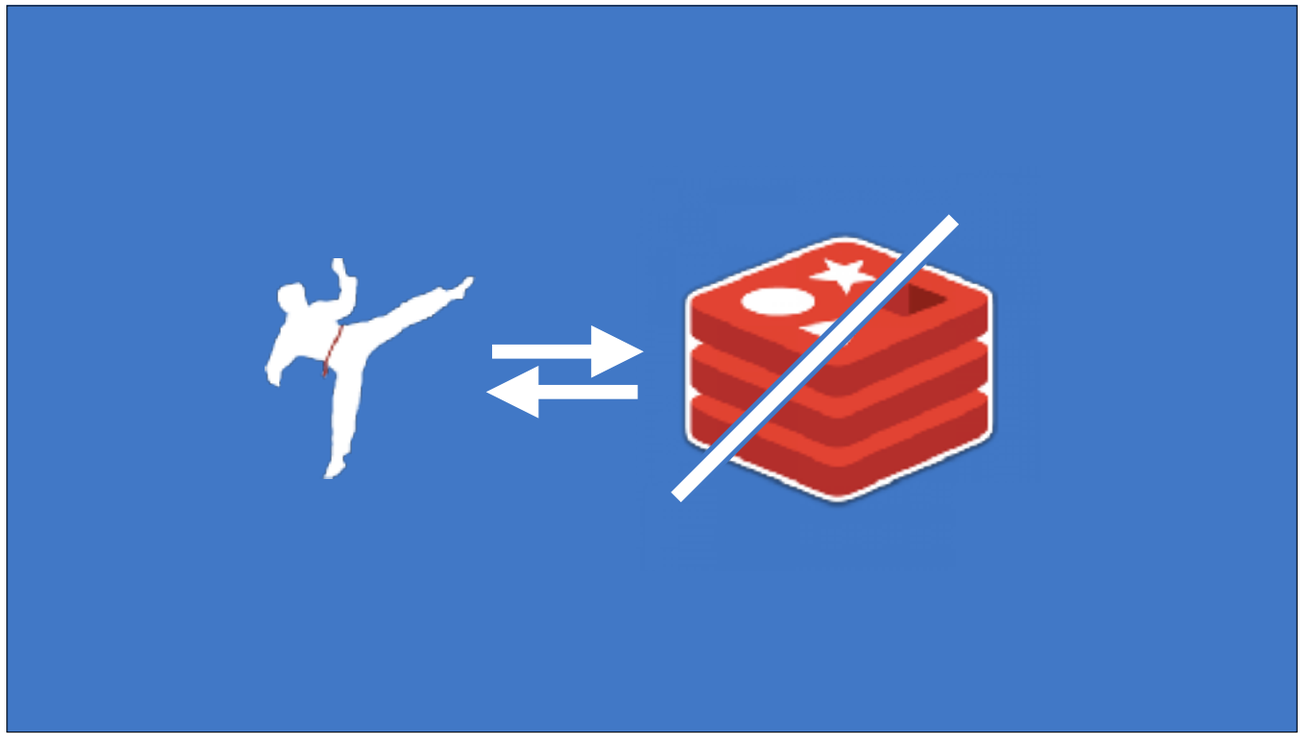


To solve those issues Sidekiq uses Redis as a backend (database).

The queue is stored as a sorted set of JSON objects. It’s quite easy to check the contents of the queue.



And since there is one central database of all the jobs, we can have multiple instances consume the same job queue.



But what happens if Redis fails? If configured to persist all changes to disk and keep a journal, nothing will happen. The second that Redis comes back online, most likely everything will be working as normal. But this mechanisms come at a cost!

RDB disadvantages

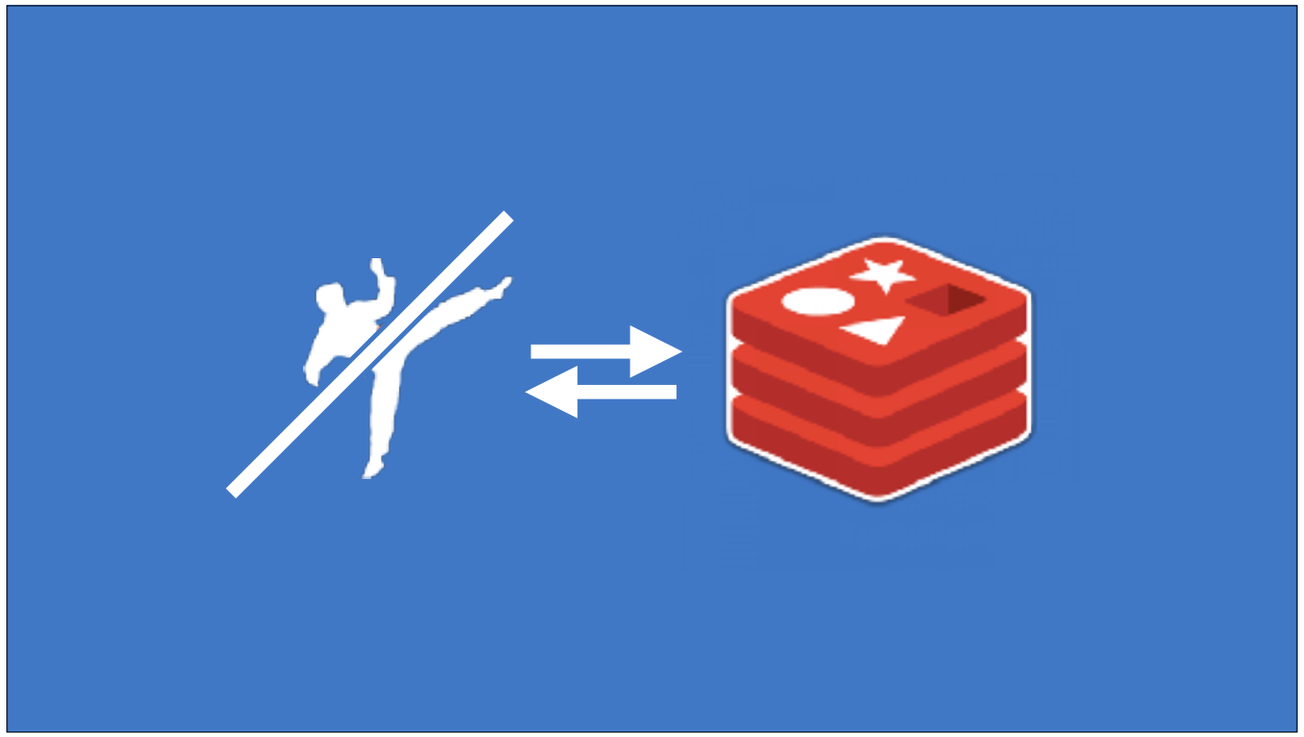
- RDB is NOT good if you need to minimize the chance of data loss in case Redis stops working (for example after a power outage). You can configure different *save points* where an RDB is produced (for instance after at least five minutes and 100 writes against the data set, but you can have multiple save points). However you'll usually create an RDB snapshot every five minutes or more, so in case of Redis stopping working without a correct shutdown for any reason you should be prepared to lose the latest minutes of data.
- RDB needs to fork() often in order to persist on disk using a child process. Fork() can be time consuming if the dataset is big, and may result in Redis to stop serving clients for some millisecond or even for one second if the dataset is very big and the CPU performance not great. AOF also needs to fork() but you can tune how often you want to rewrite your logs without any trade-off on durability.

If you opt to keep a transactional jurnal, be prepared to lose some data and to have slow connections.

AOF disadvantages

- AOF files are usually bigger than the equivalent RDB files for the same dataset.
- **AOF can be slower than RDB depending on the exact fsync policy.** In general with fsync set to *every second* performances are still very high, and with fsync disabled it should be exactly as fast as RDB even under high load. **Still RDB is able to provide more guarantees about the maximum latency even in the case of an huge write load.**
- In the past we experienced rare bugs in specific commands (for instance there was one involving blocking commands like BRPOPLPUSH) causing the AOF produced to not reproduce exactly the same dataset on reloading. This bugs are rare and we have tests in the test suite creating random complex datasets automatically and reloading them to check everything is ok, but this kind of bugs are almost impossible with RDB persistence. To make this point more clear: the Redis AOF works incrementally updating an existing state, like MySQL or MongoDB does, while the RDB snapshotting creates everything from scratch again and again, that is conceptually more robust. However - 1) It should be noted that every time the AOF is rewritten by Redis it is recreated from scratch starting from the actual data contained in the data set, making resistance to bugs stronger compared to an always appending AOF file (or one rewritten reading the old AOF instead of reading the data in memory). 2) We never had a single report from users about an AOF corruption that was detected in the real world.

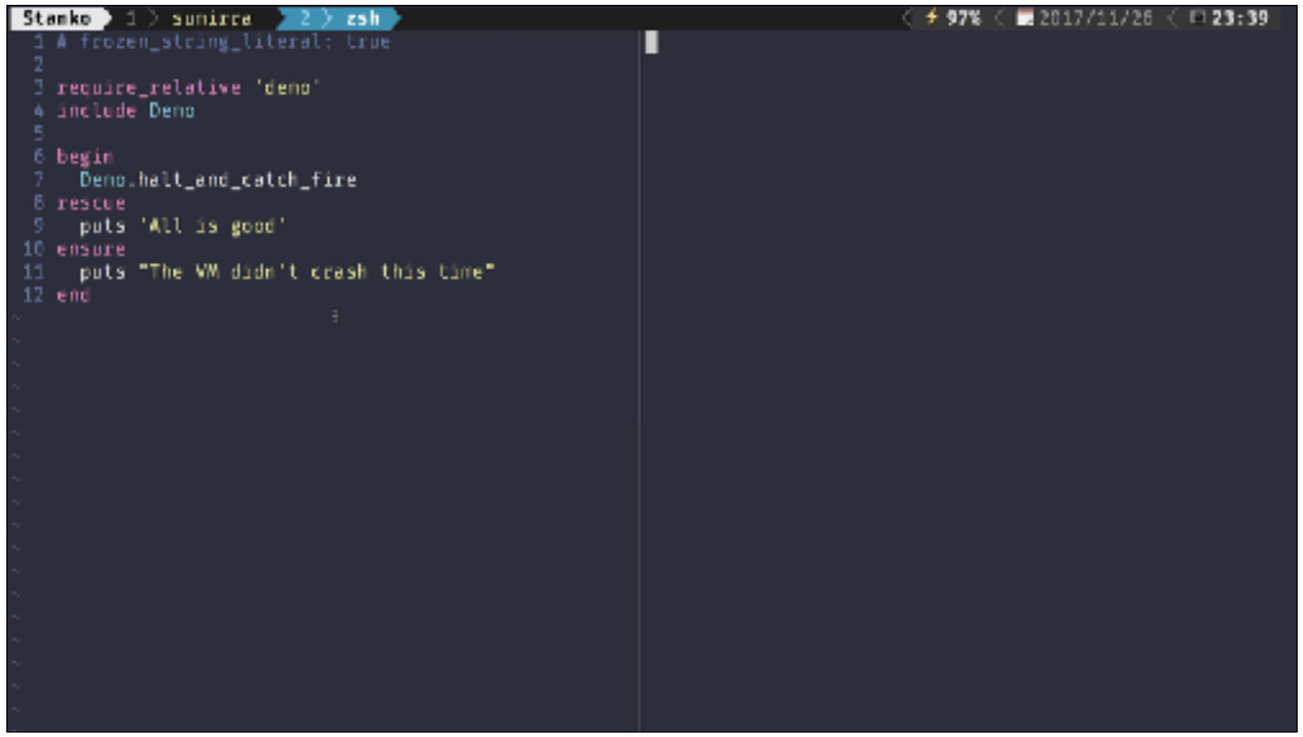
And if you opt for data persistence on disk be prepared to experience performance fluctuations.



Now we come to the real issues. What happens if your worker fails?

```ruby
class TestWorker
  include Sidekiq::Worker

  def perform(*args)
    # Do something
  rescue
    # Do something else
  end
end
```

Well… We can always use Ruby to resolve exceptions! In fact, Sidekiq already does this for us! And it offers retry strategies, and exponential backoff and other bells and whistles!

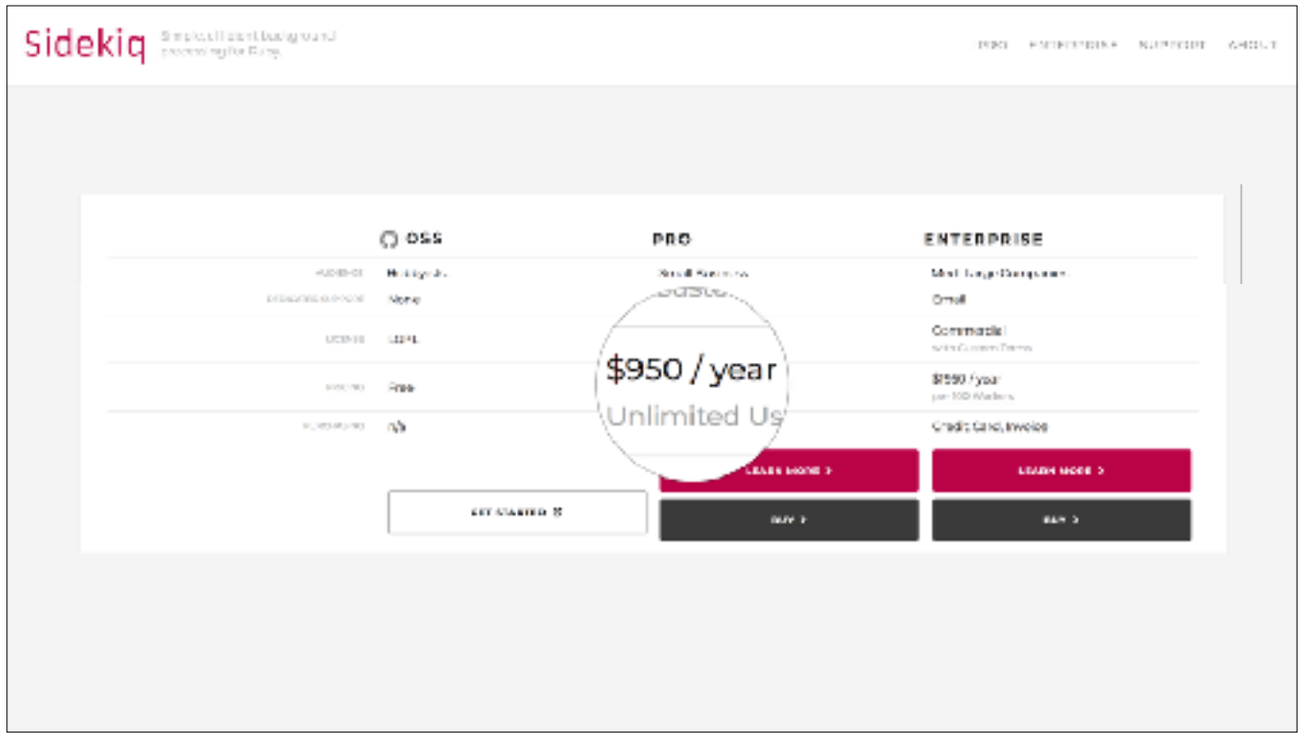


Well… No, not really. There are some exceptions that simply cant be caught.

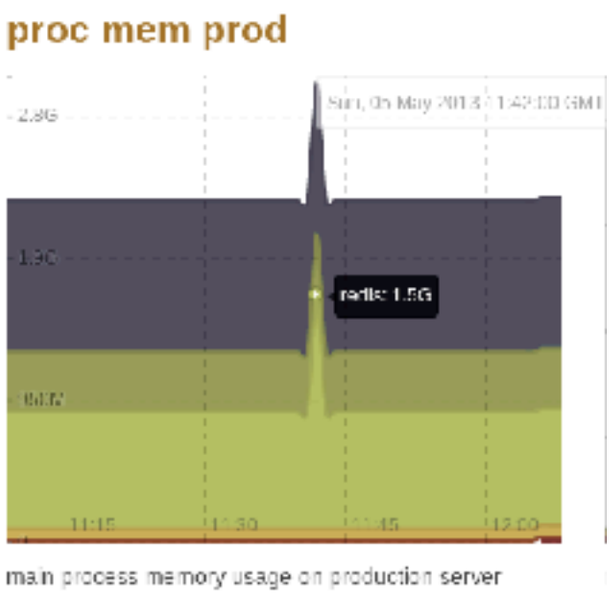
Process Crashes

If the Sidekiq process segfaults or crashes the Ruby VM, any jobs that were being processed are lost. Sidekiq Pro offers a reliable queueing feature which does not lose those jobs.

Note that Sidekiq does offer protection against those kinds of errors, though it doesn’t specify how it does so.



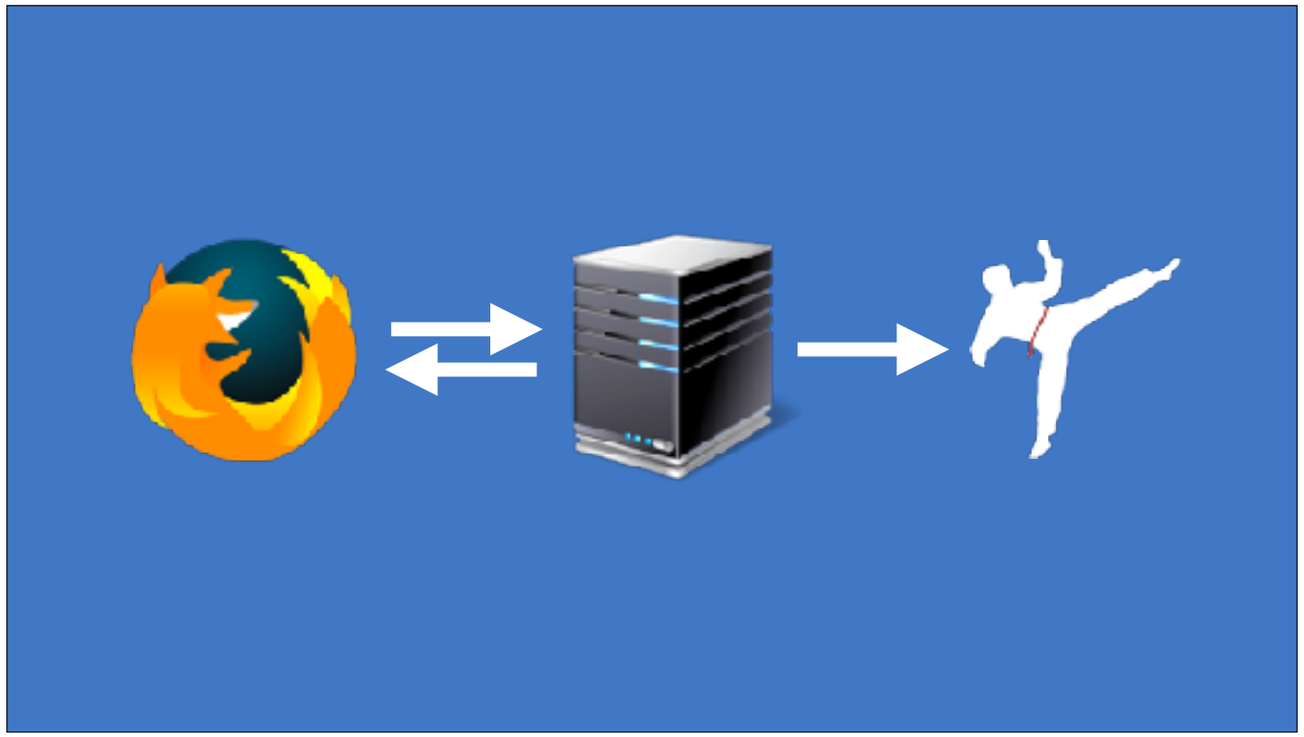
And that feature comes with a $1000 price tag. But even the $2000 version won’t save you from the following.



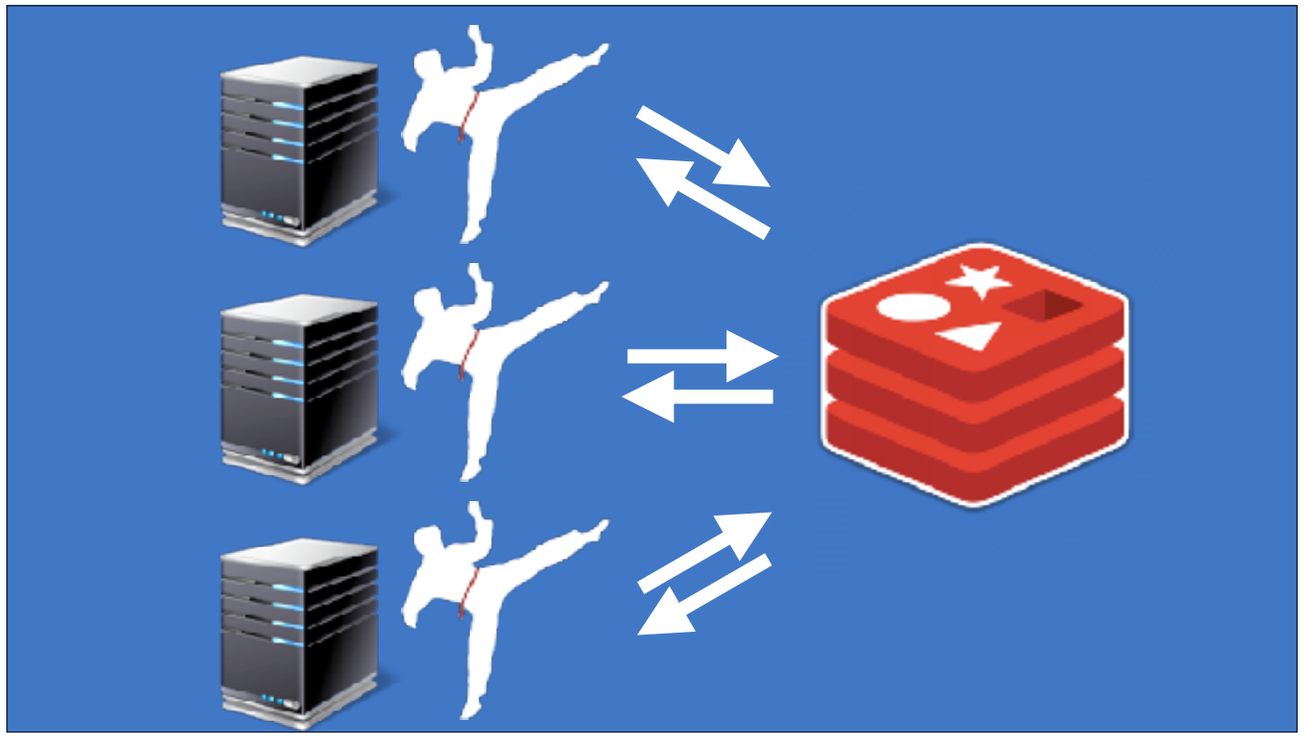
main process memory usage on production server

Memory consumption! Redis is an in-memory database, which means that it keeps all data in-memory. If you pass it 1.5GB of data, it will store that date in-memory. Even with persistence turned on, everything is still stored in-memory.

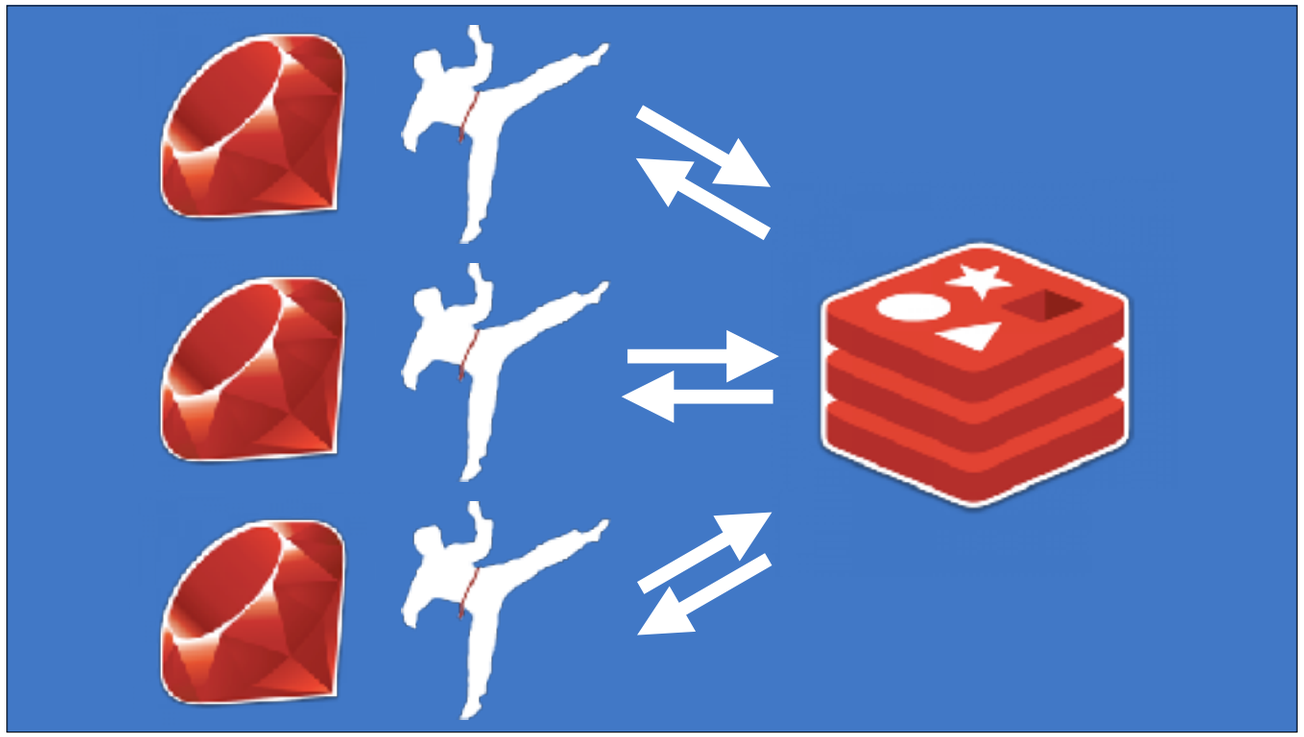
Ok, that can be attributed to bad design. Somebody is passing whole objects instead of only their IDs. In that case we have a potential scaling problem.



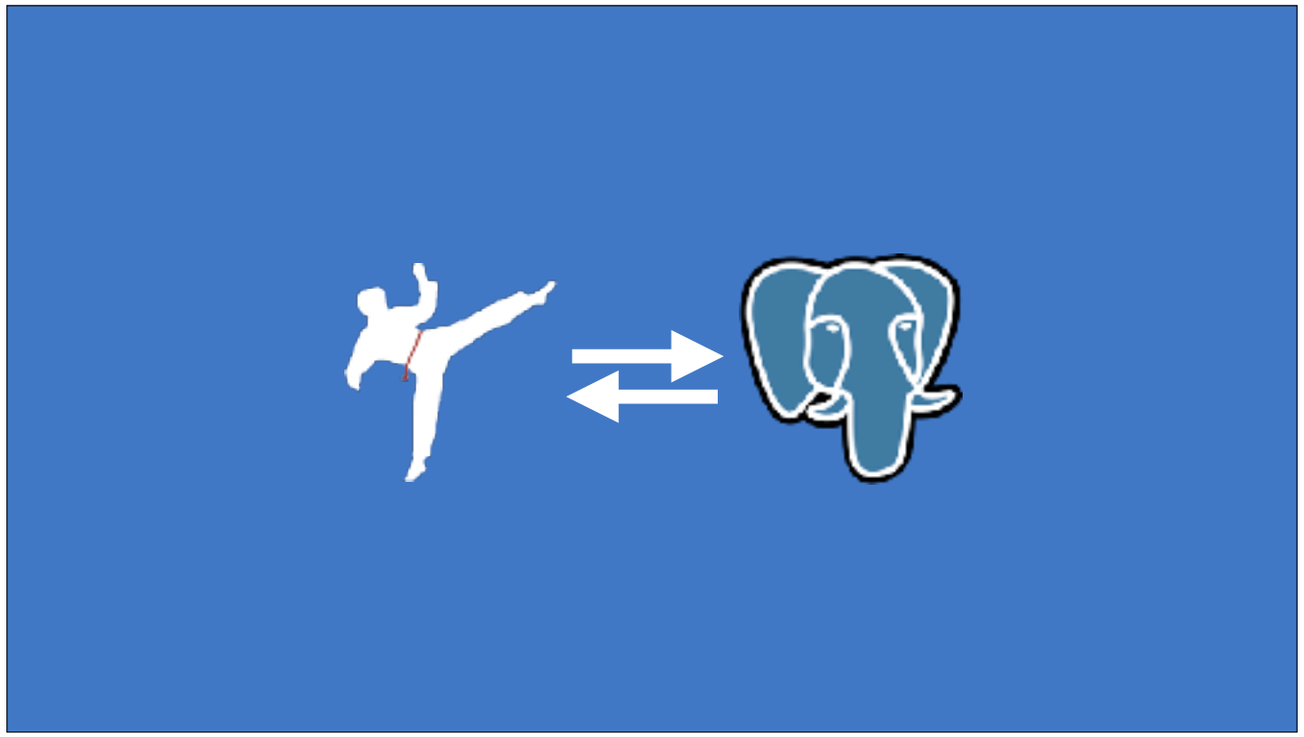
While our web app seemingly talks to a single Sidekiq instance.



Each pedis worker keeps an instance of the whole application in memory. This is the most common case I have seen while working with Rails applications.



To solve this problem we can start Sidekiq workers as separate light-weight processes. This would solve the memory consumption issue.



But do we then give the workers direct access to the database? If yes then we need to manage models in two separate apps.

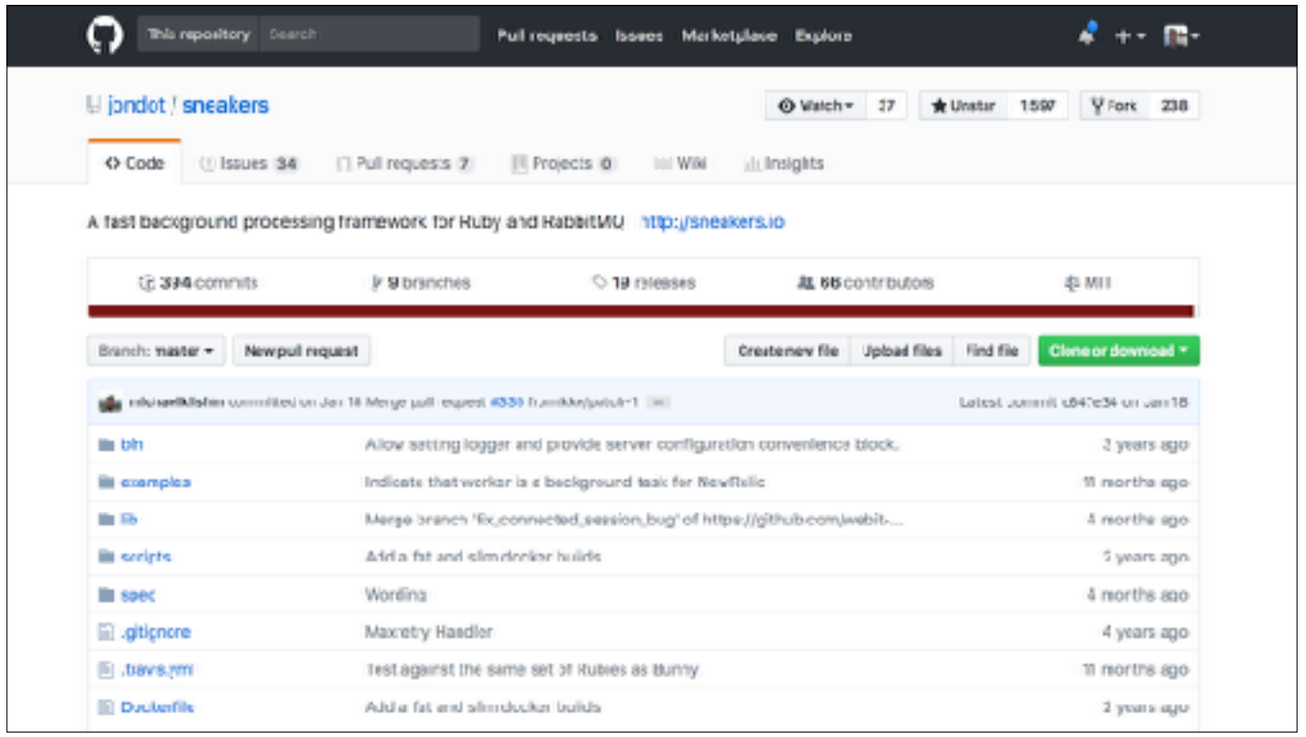
If no then we need to store and retrieve data through our own app, thus increasing load and potentially introducing temporary data do our DB.

02

RABBITMQ

This brings us to RabbitMQ.

RabbitMQ is a lot more than a simple job queue, but I'll start with that utility of it, since it's the one directly comparable to Sidekiq.



To use it as a background worker I would highly recommend the Sneakers project.

```ruby
class TitleScraper
  include Sneakers::Worker

  from_queue 'downloads'

  def work(msg)
    title = extract_title(msg)
    logger.info "FOUND <#{title}>"
    ack!
  end

  private

  def extract_title(html)
    html.scan(/<title>(.*?)<\/title>/).flatten.first
  end
end
```

```ruby
class TestWorker
  include Sidekiq::Worker

  def perform(*args)
    # Do something
  end
end
```

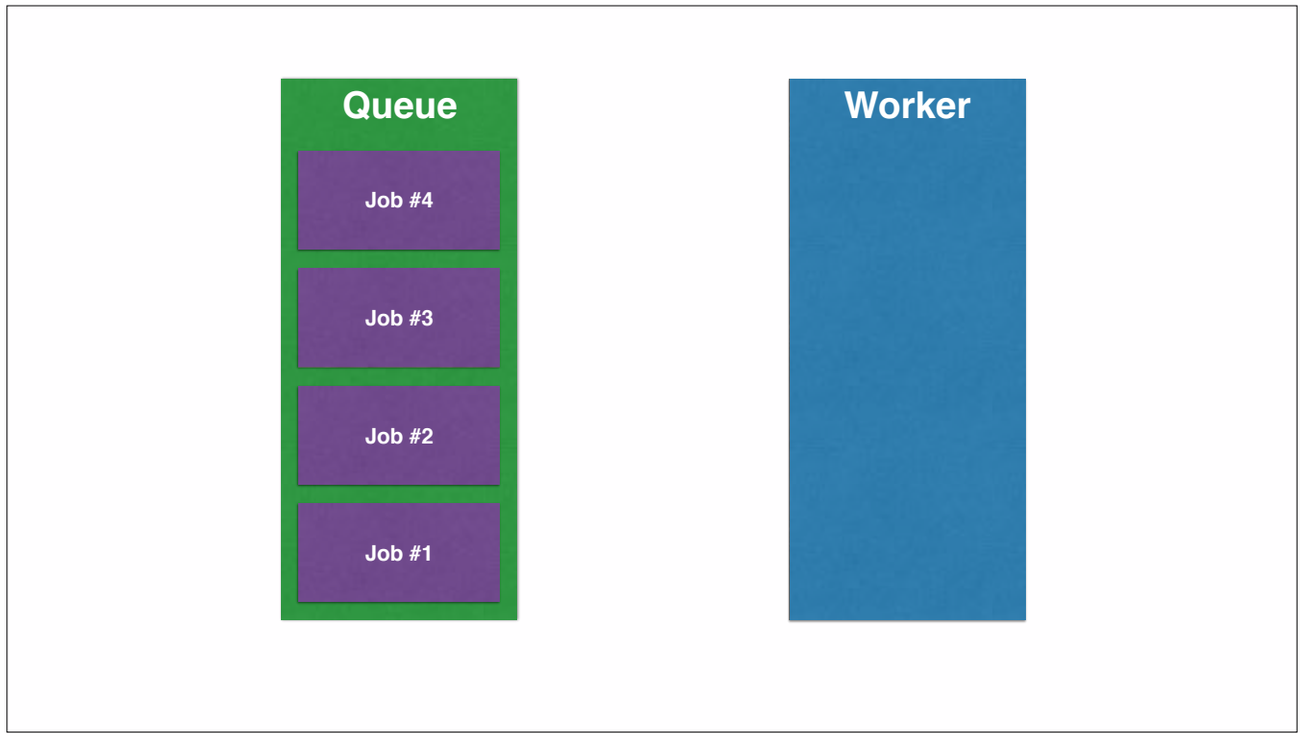
You would use it exactly the same as you would use Sidekiq. Even the syntax is similar. On the left is a Sneakers worker, on the right is a Sidekiq worker.

```ruby
class TitleScraper
  include Sneakers::Worker

  from_queue 'downloads'

  def work(msg)
    title = extract_title(msg)
    logger.info "FOUND <#{title}>"
    ack!
  end

  private

  def extract_title(html)
    html.scan(/<title>(.*?)<\/title>/).flatten.first
  end
end
```

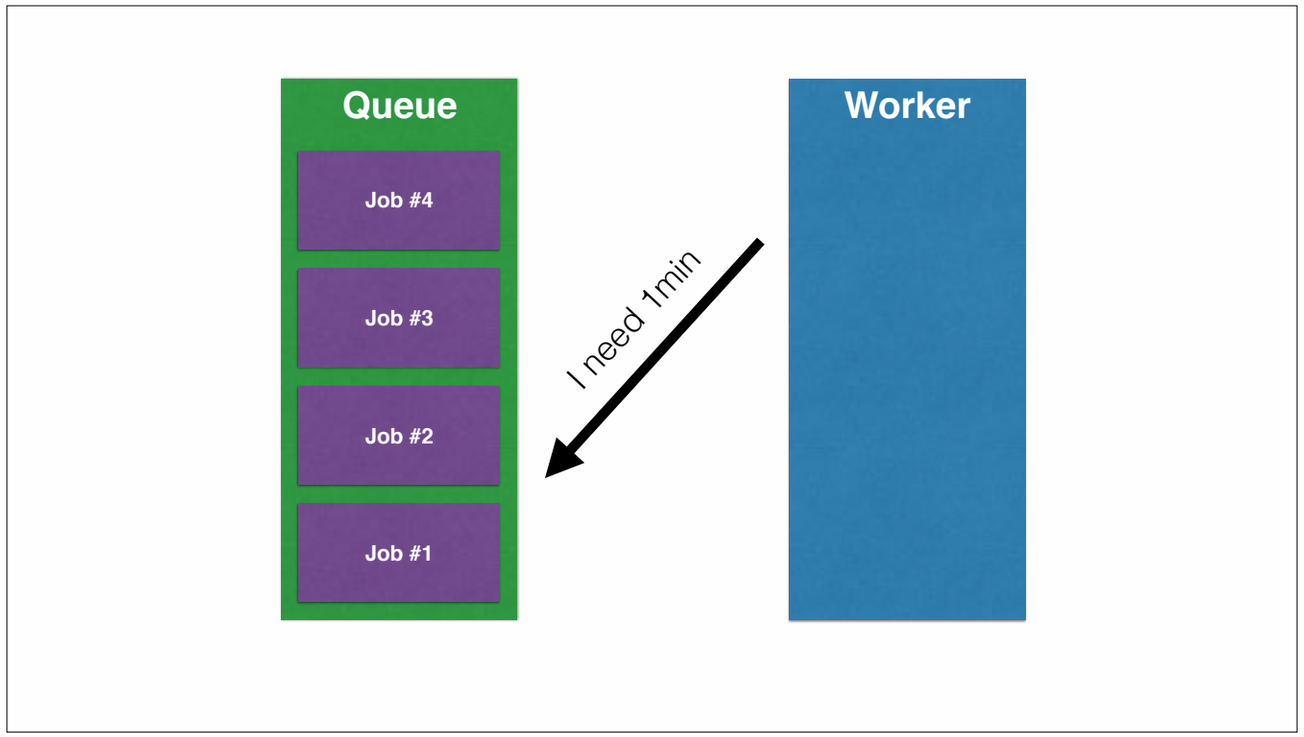
Here we can also notice a big feature of RabbitMQ and it’s underlaying protocol AMPQ

```ruby
class TitleScraper
  include Sneakers::Worker

  from_queue 'downloads'

  def work(msg)
    title = extract_title(msg)
    logger.info "FOUND <#{title}>"
    ack!
  end

  private

  def extract_title(html)
    html.scan(/<title>(.*?)<\/title>/).flatten.first
  end
end
```

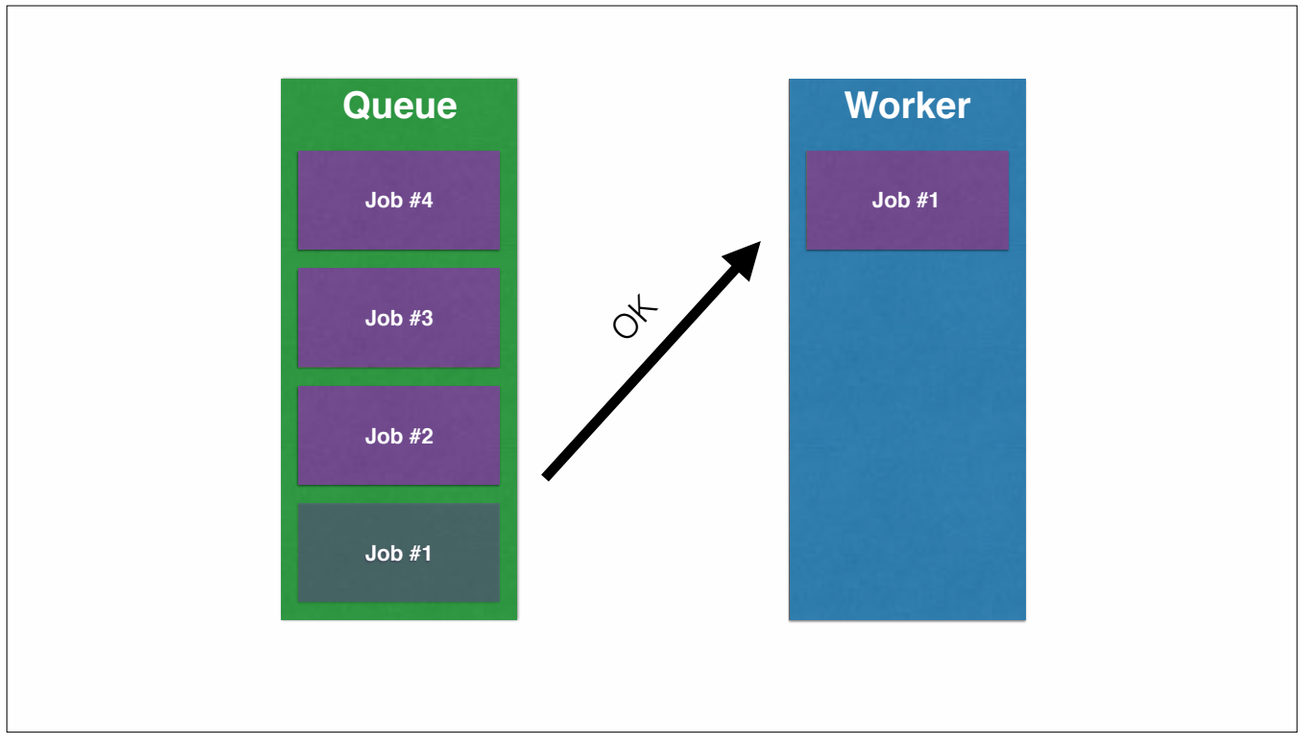
This little `ack!` method guarantees that the job will be processed no matter if RabitMQ or the worker fail.



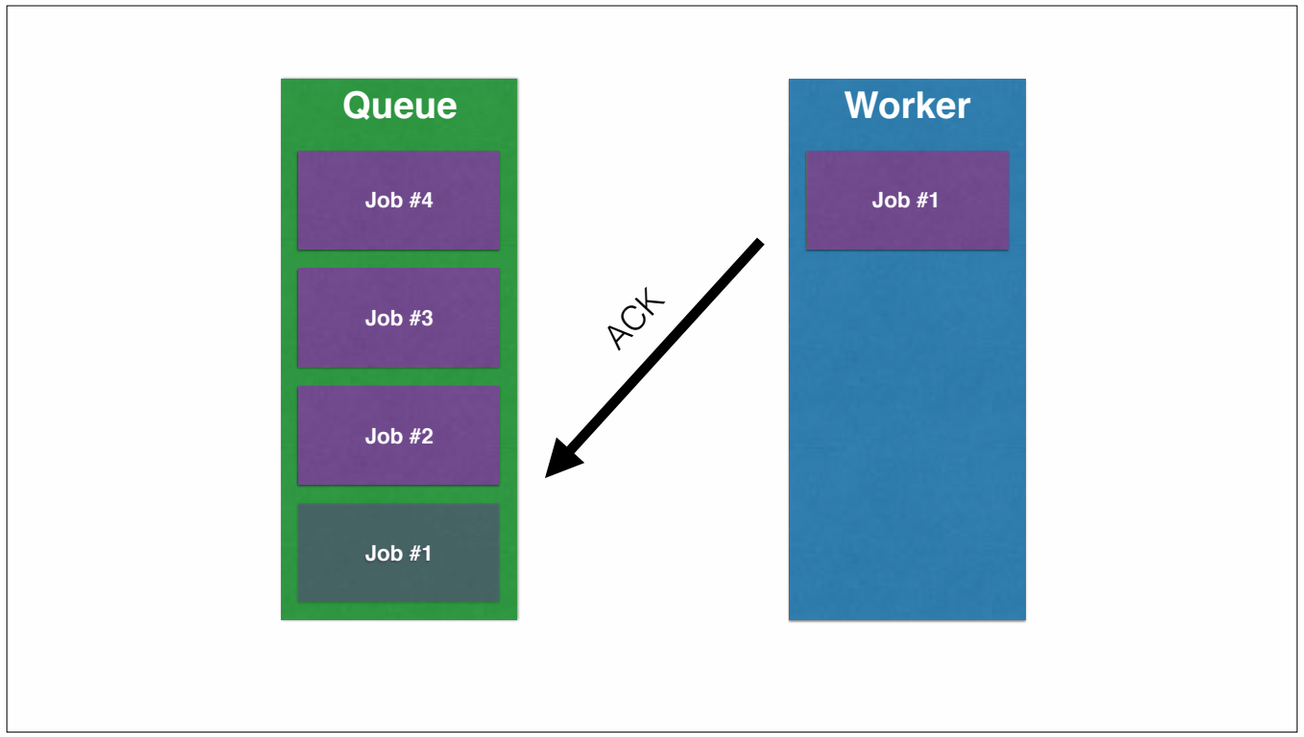
AMPQ has two ways of fetching data from a queue. One is ACK mode and the other is NO-ACK mode. Here is an example of how ACK mode works.



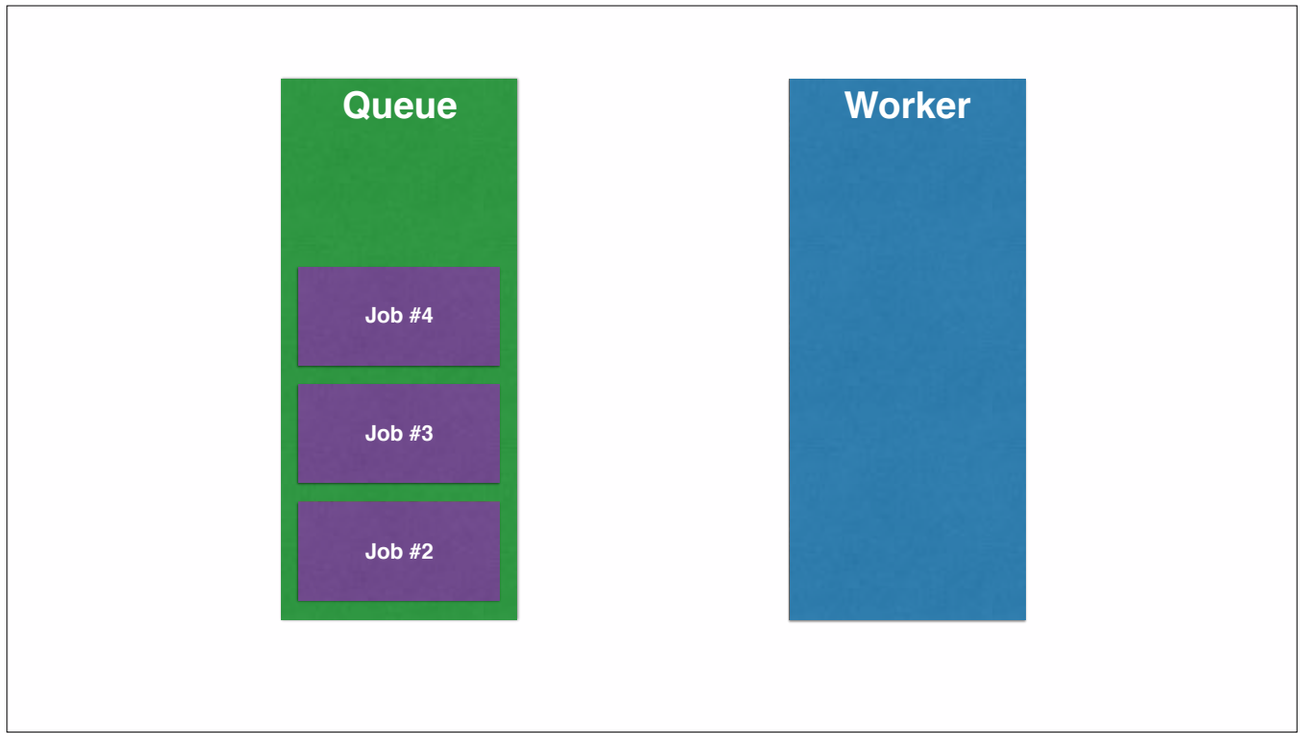
A worker, before poping a queue first specifies how long it will take to process the job. In this case 1min.



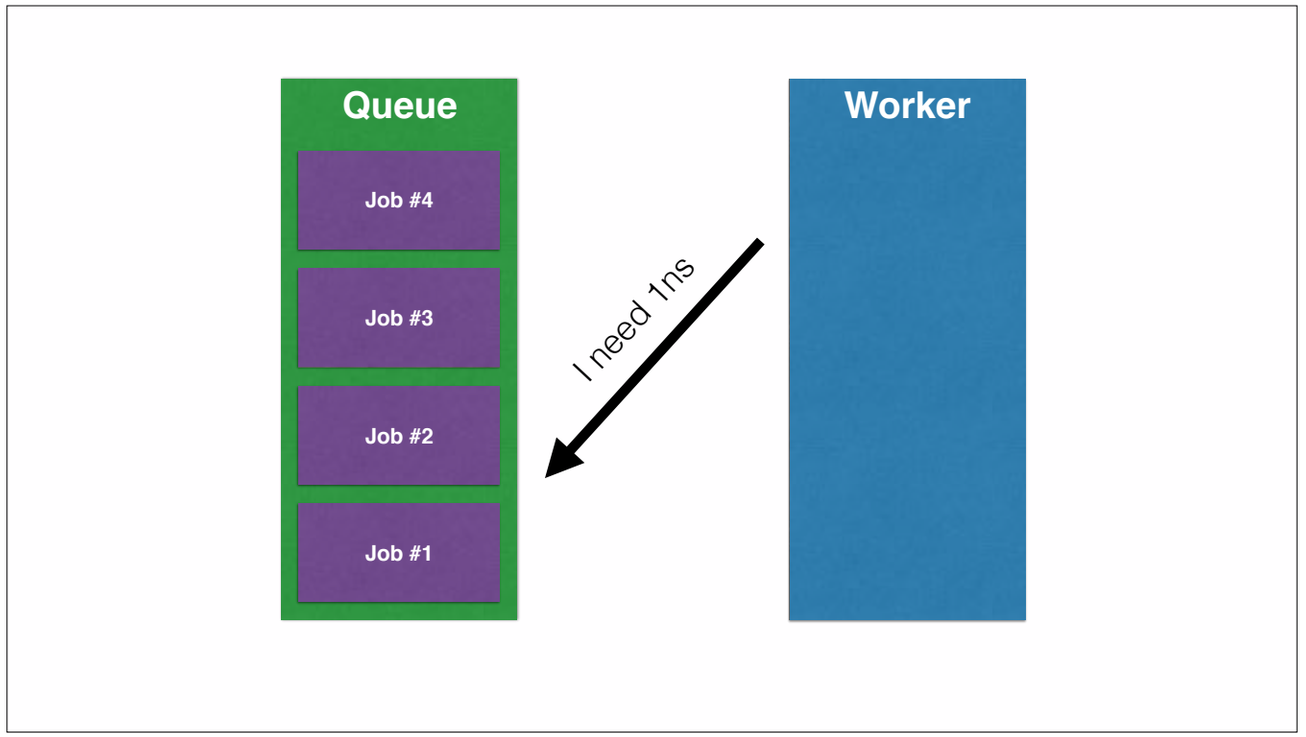
At that point the job is virtually removed from the queue, and the worker starts processing it. This job is still in the queue, but other workers can’t see it!



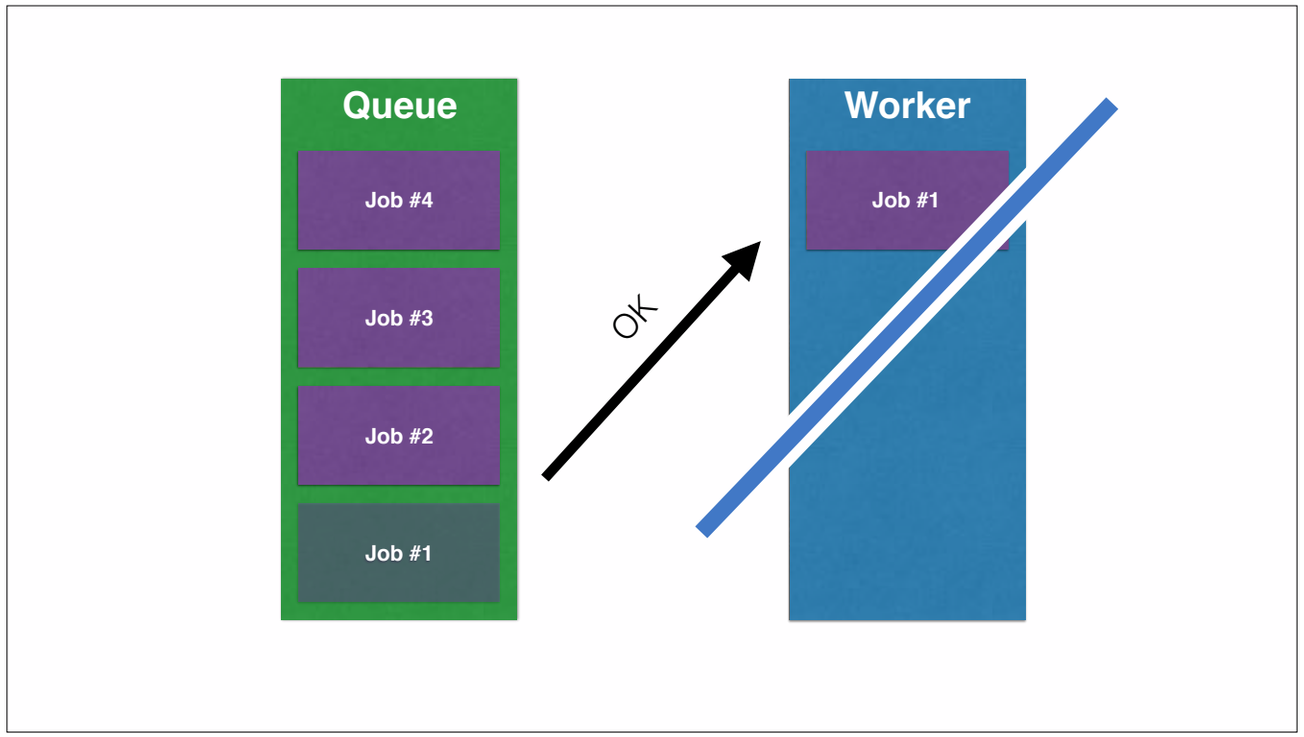
When the worker is finished it sends an ACK signal to the queue to indicate it has finished



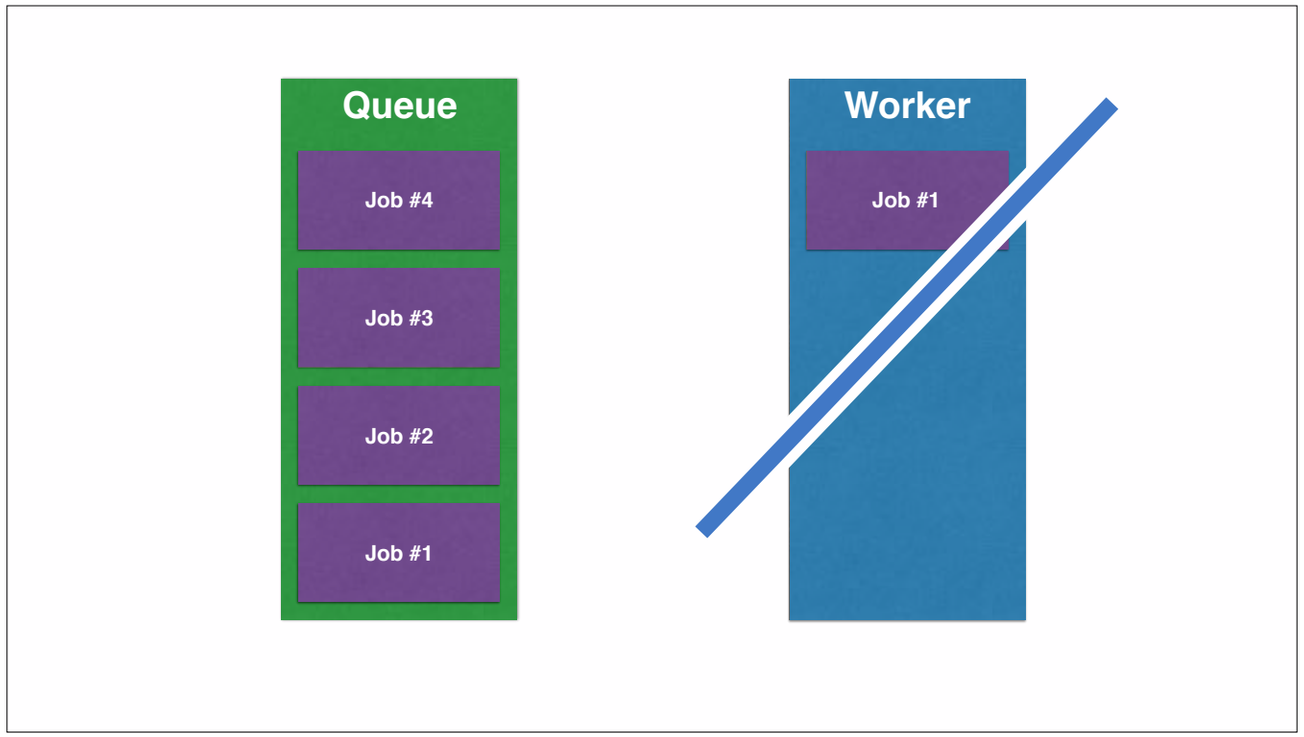
At that point the job is removed from the queue



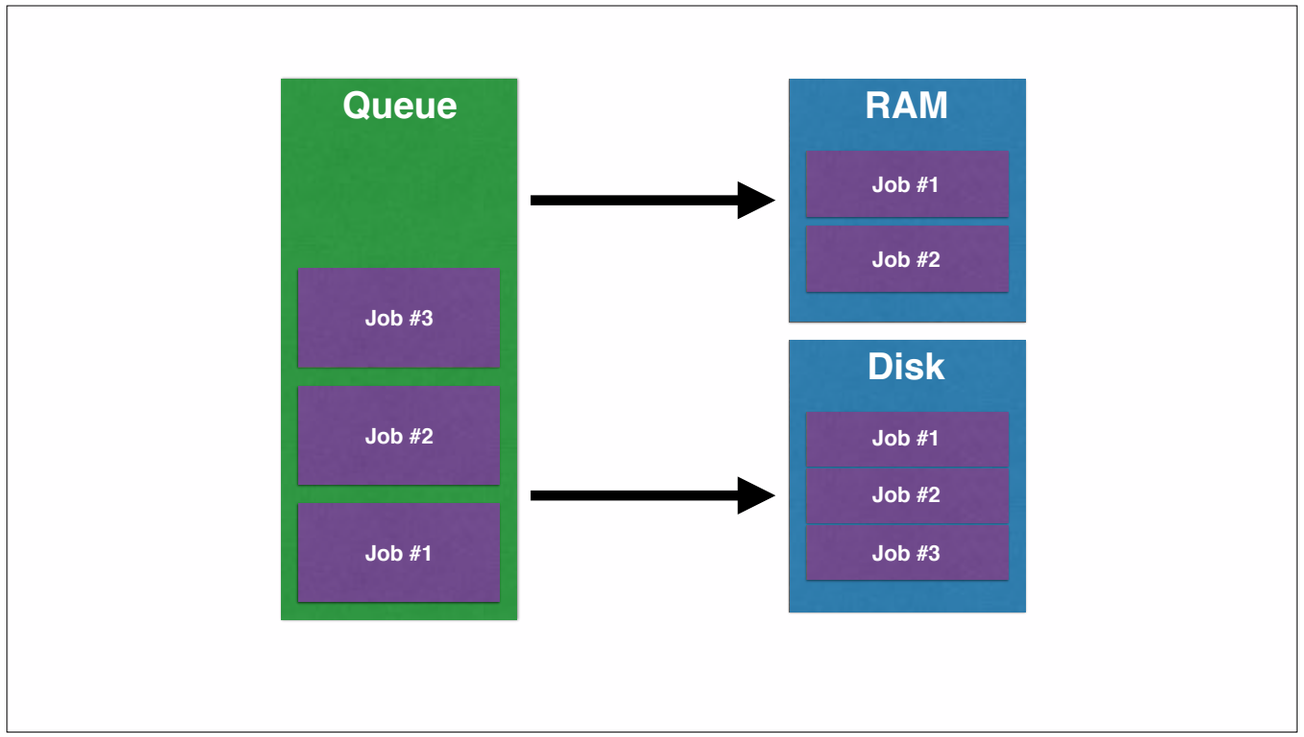
If a worker set an unrealistic goal, like in this case 1ns



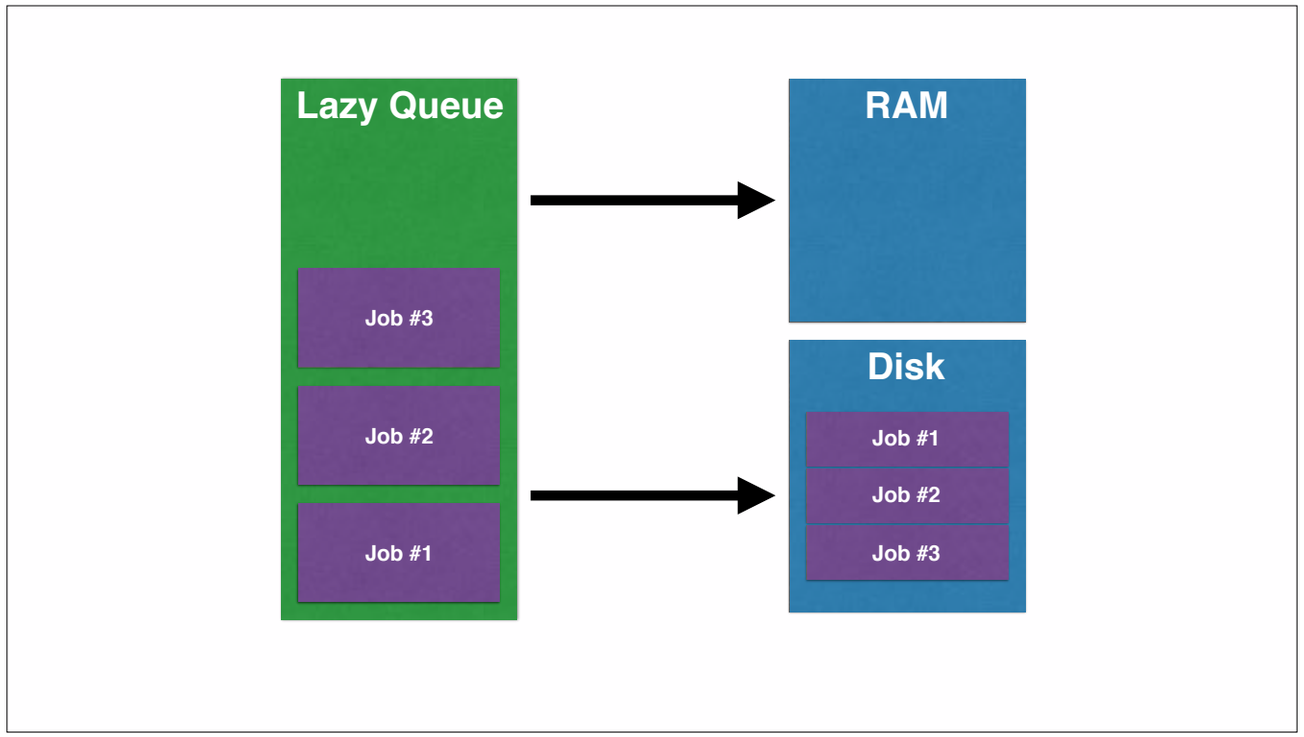
Or if a worker fails while processing



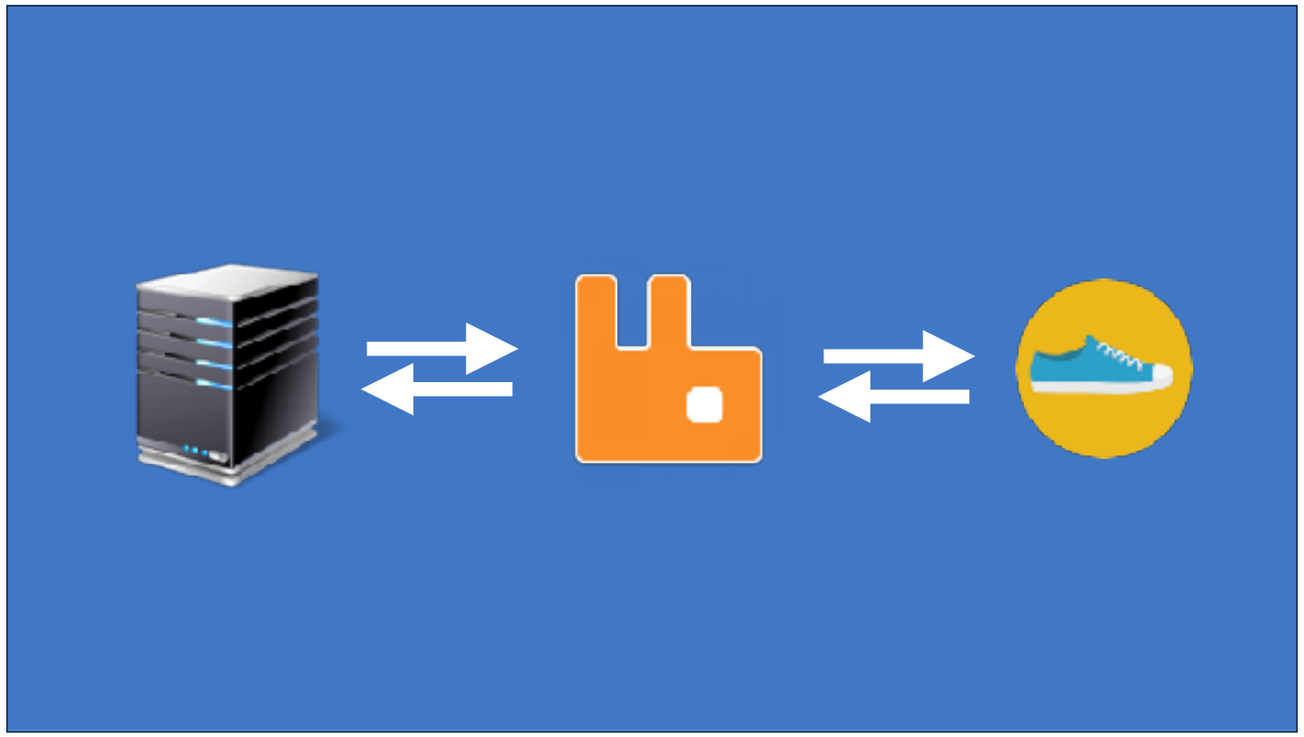
As soon as the timer runs out, the job is put back in the queue. No data is lost! Even if RabbitMQ restarts!



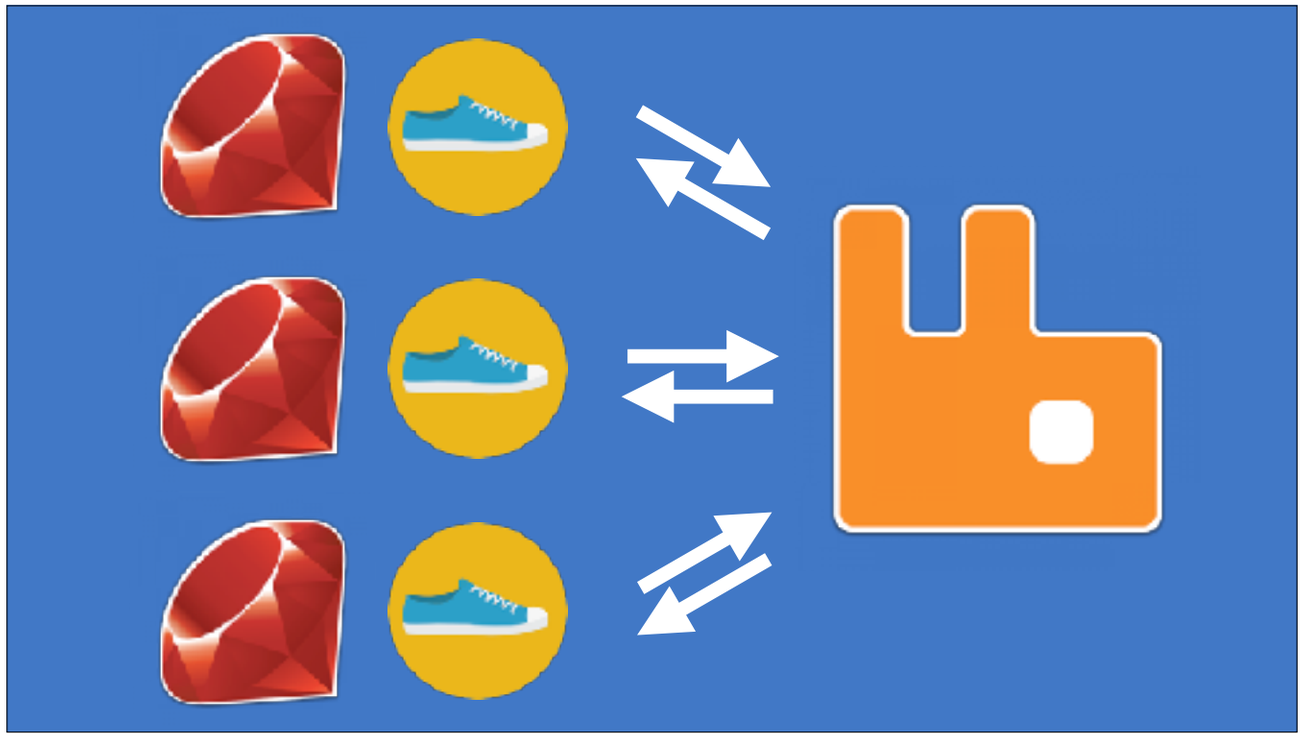
RabbitMQ always persists data to the disk in a transactional fashion, similar to Postgres. And it keeps as many jobs as it can in memory. If under memory pressure, it will start utilising the disk more and more.



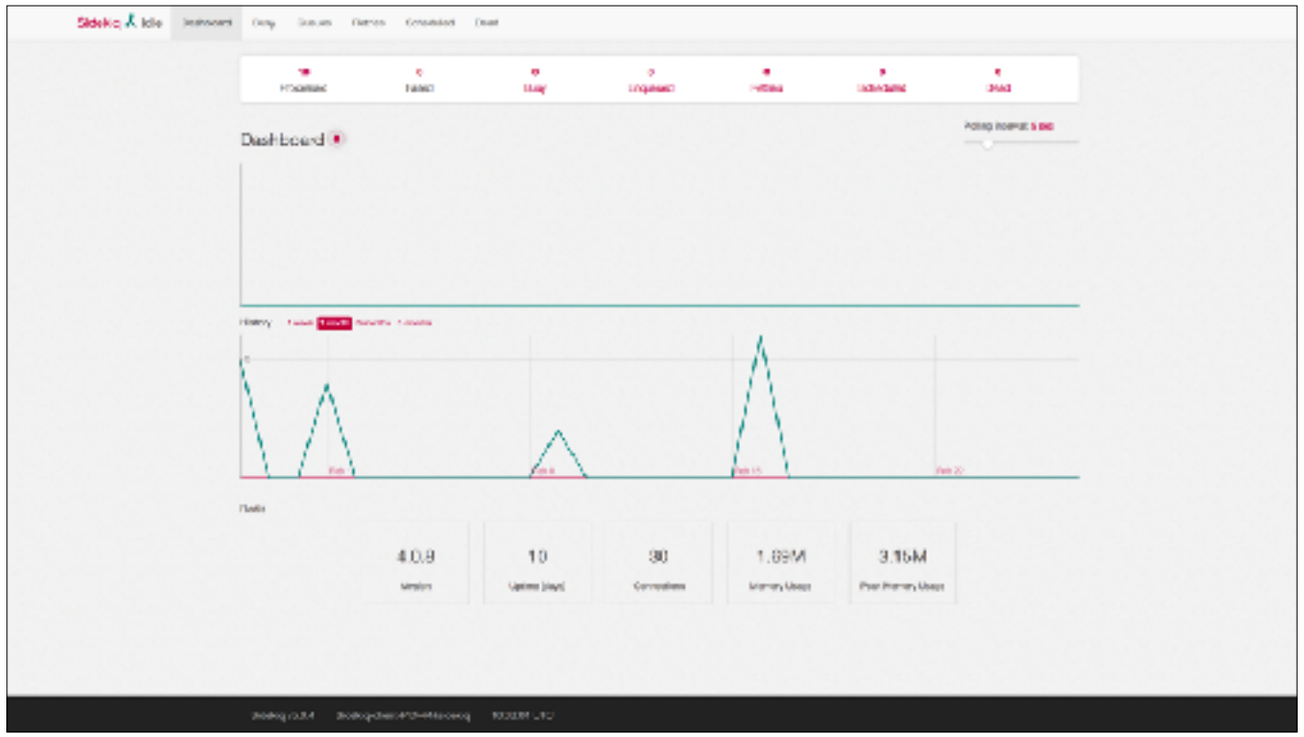
There are also other kinds of queues, like Lazy Queues, which keep everything on disk. They are intended to be used with large payloads.



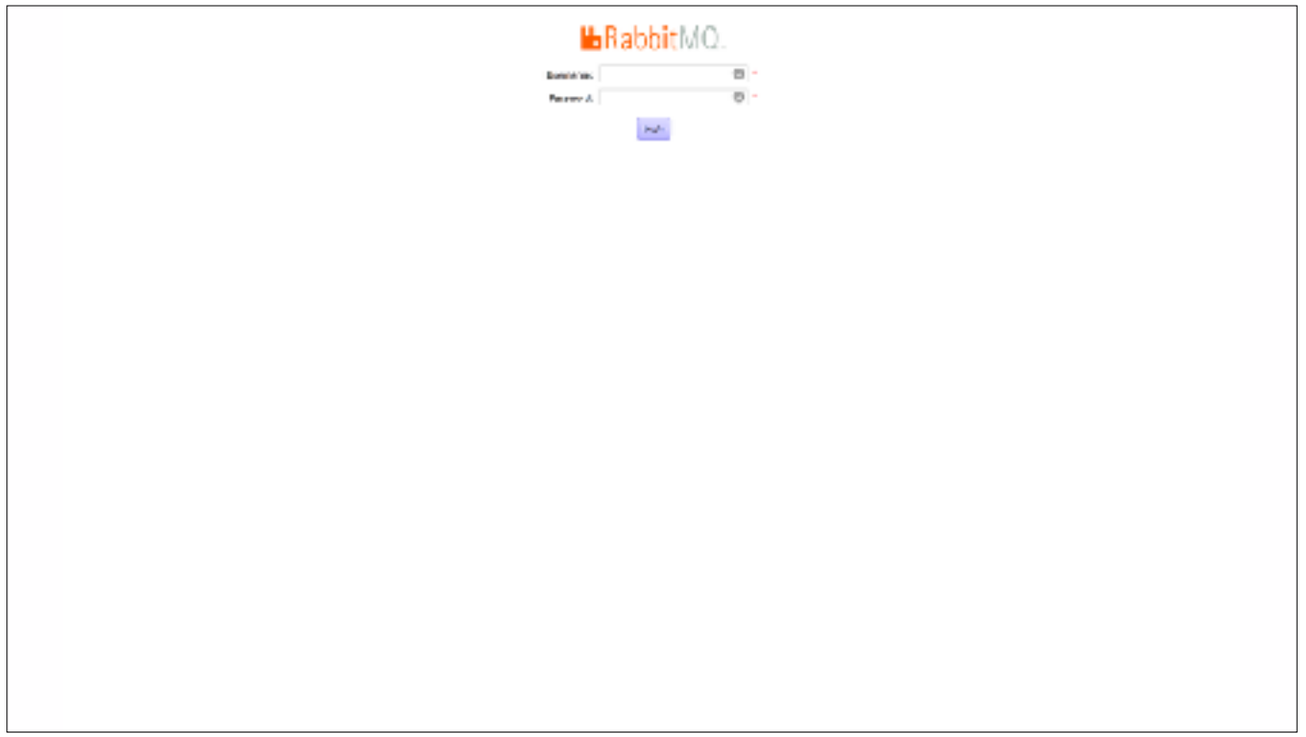
Back to our example. Since we are now able to transport basically arbitrary sizes of data through our queue we don’t need access to the database anymore, and we don’t need to ask our main app for data it can just give us all the information we need to process a job.



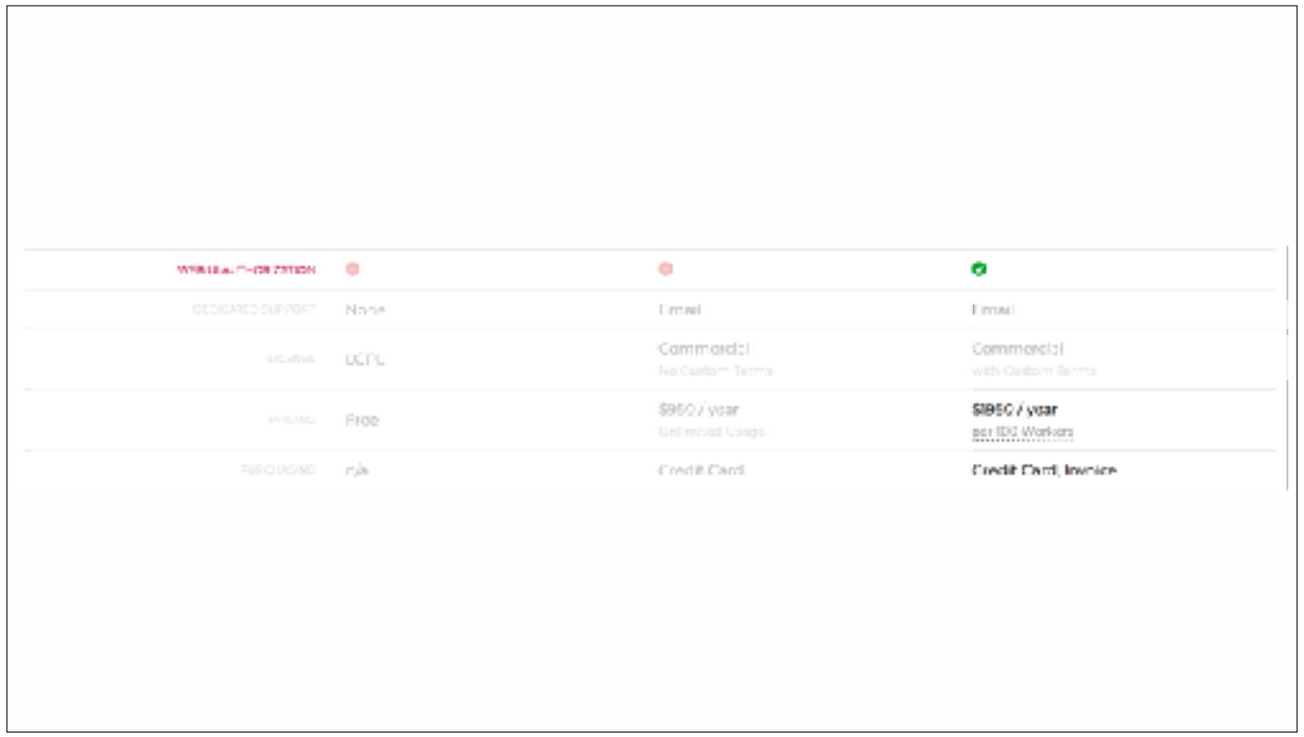
And since now we can write our workers as lightweight Ruby processes, memory consumption also goes down, and we can easily scale those processes!



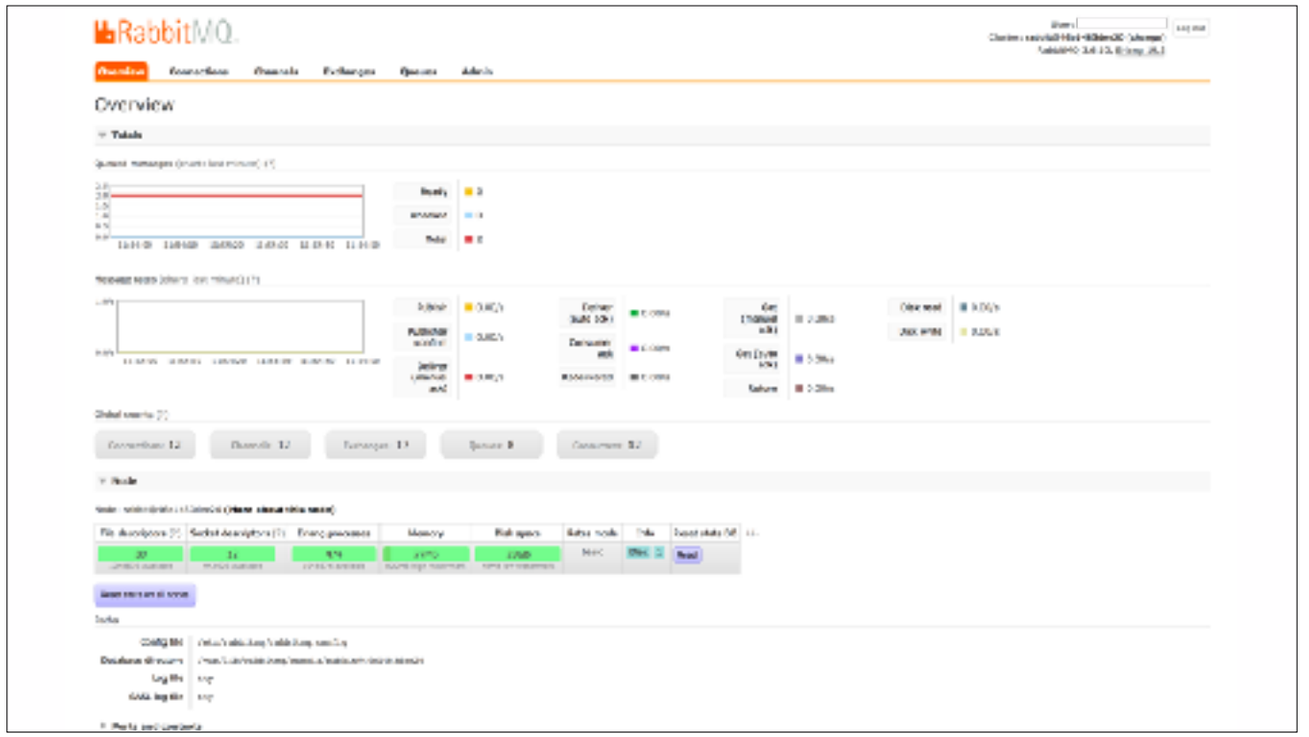
To be honest Sidekiq comes with a cool interface to manage and track how your jobs progress.



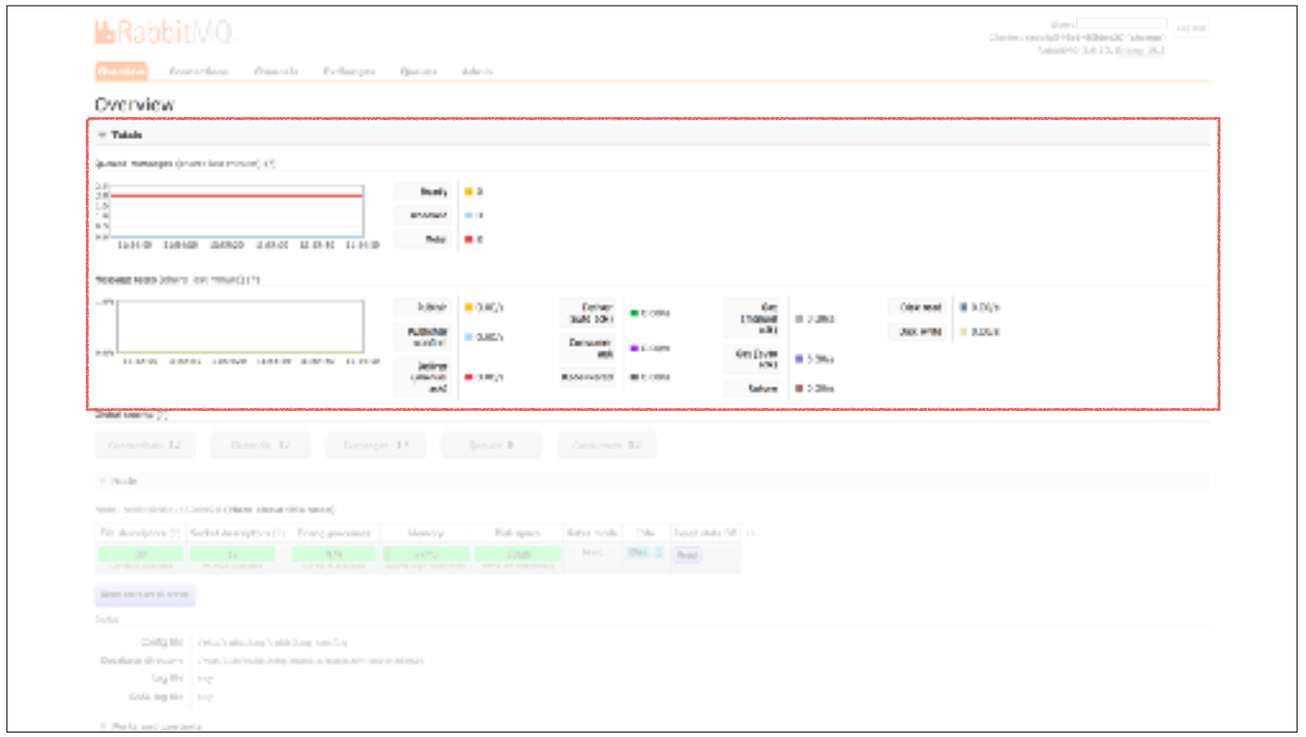
RabbitMQ also has a nice UI. In my opinion it’s biggest feature is that it’s free.



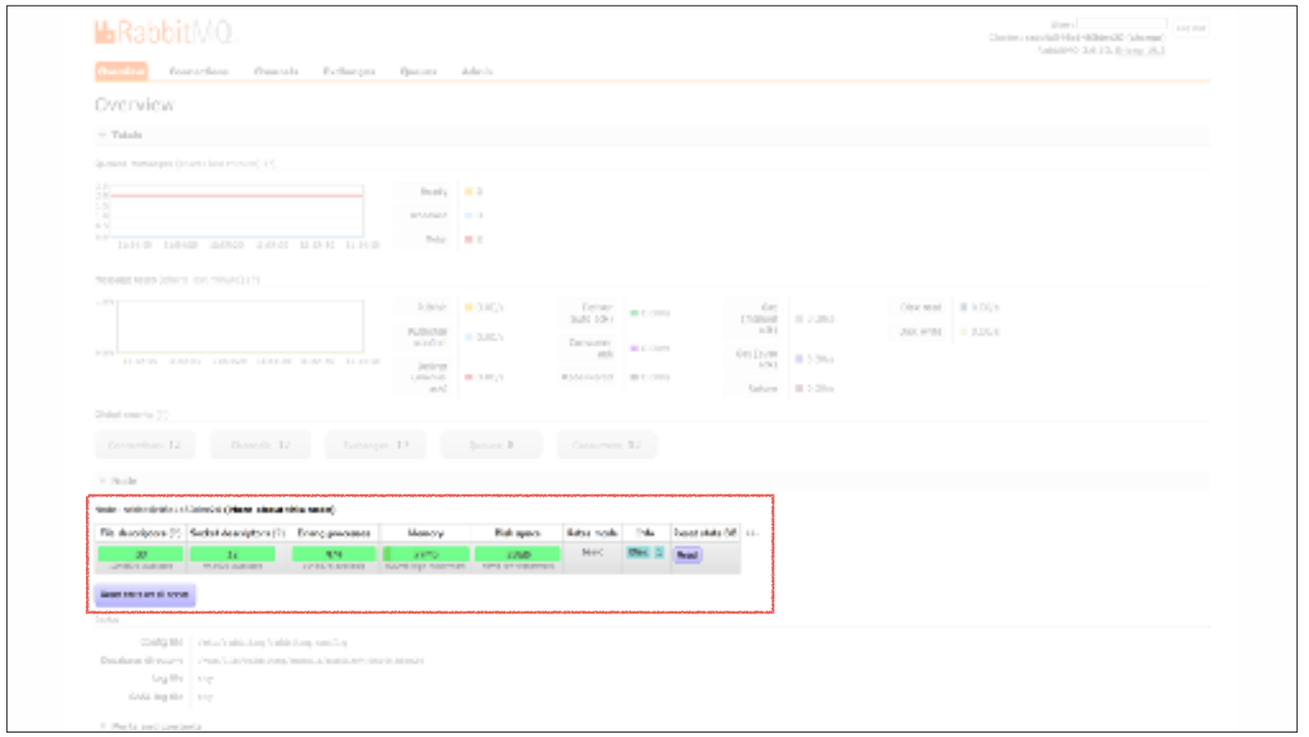
And it comes with authorisation, that’s a hell of a lot better than $2k for a simple login screen.



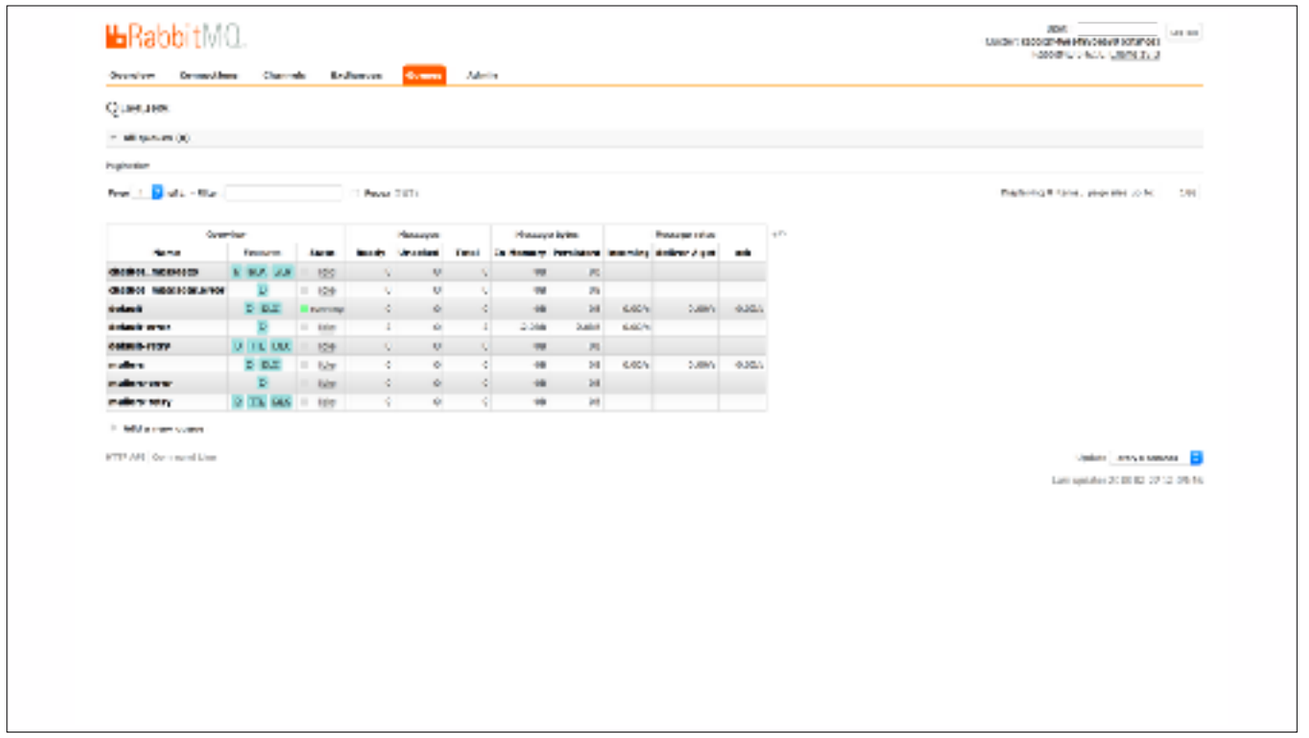
Anyway, you have the standard overview screen.



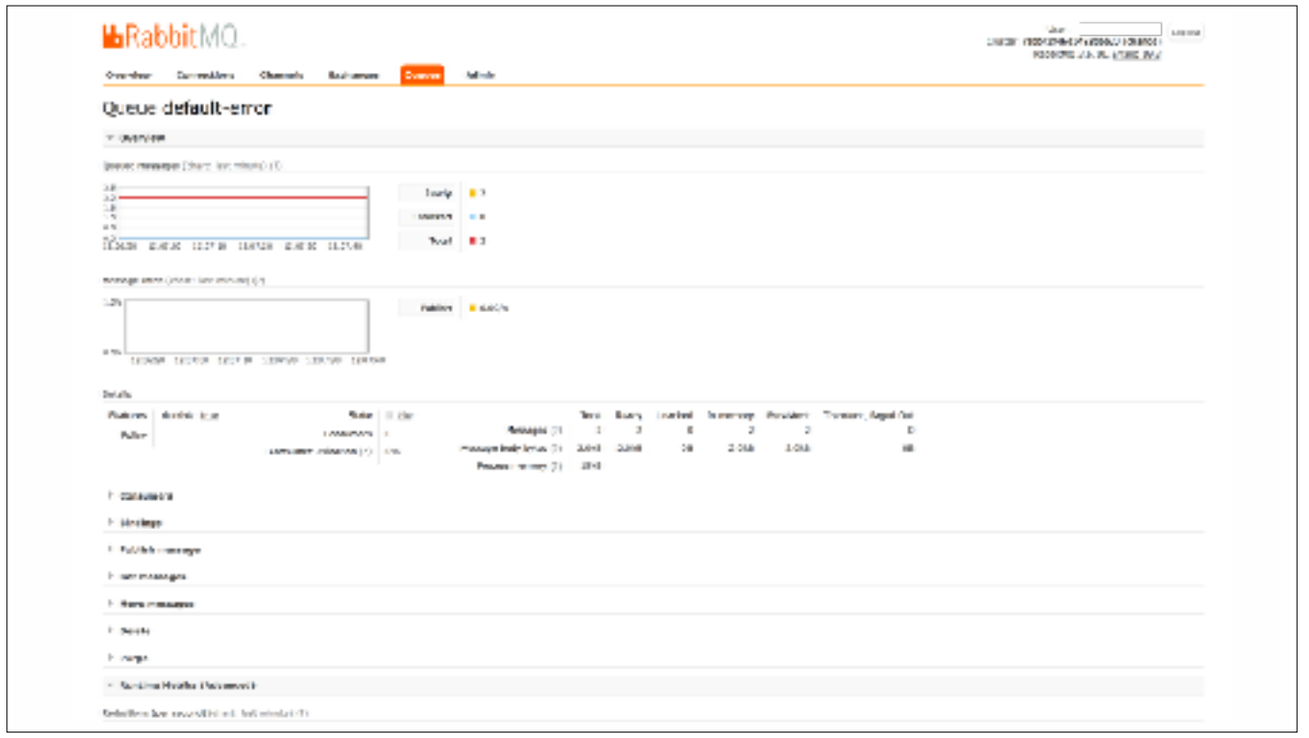
It show how many messages are being processed.



And how the systems resources are being utilized.

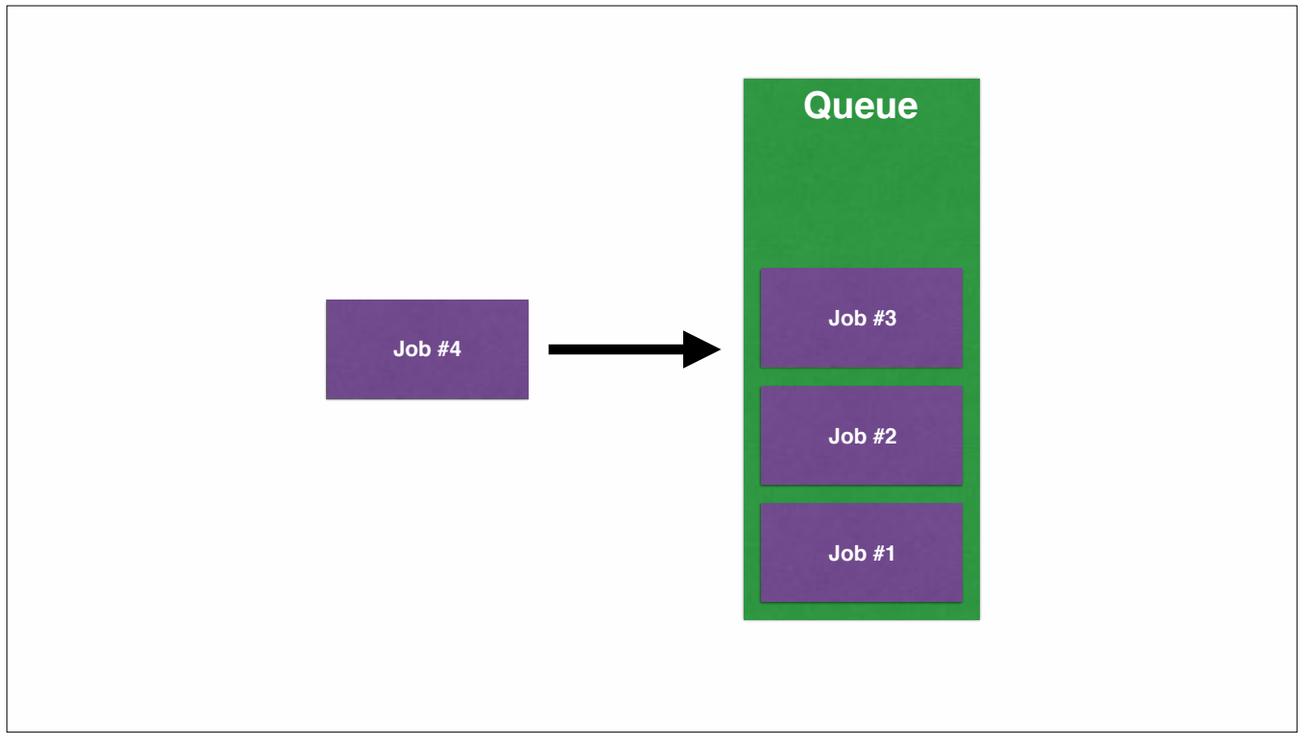


You can also manage individual queues.

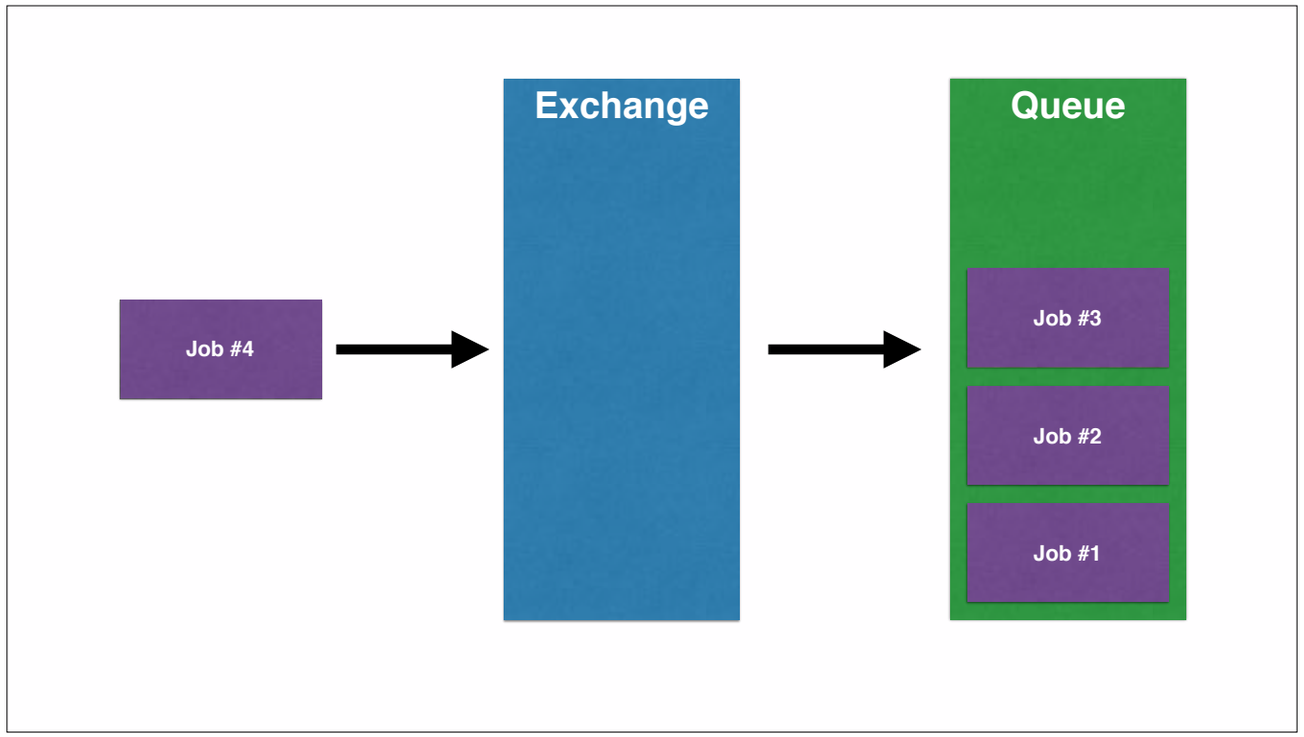


You can look at it’s individual statistics. But you can also modify it, read, create and delete messages. And a lot more.

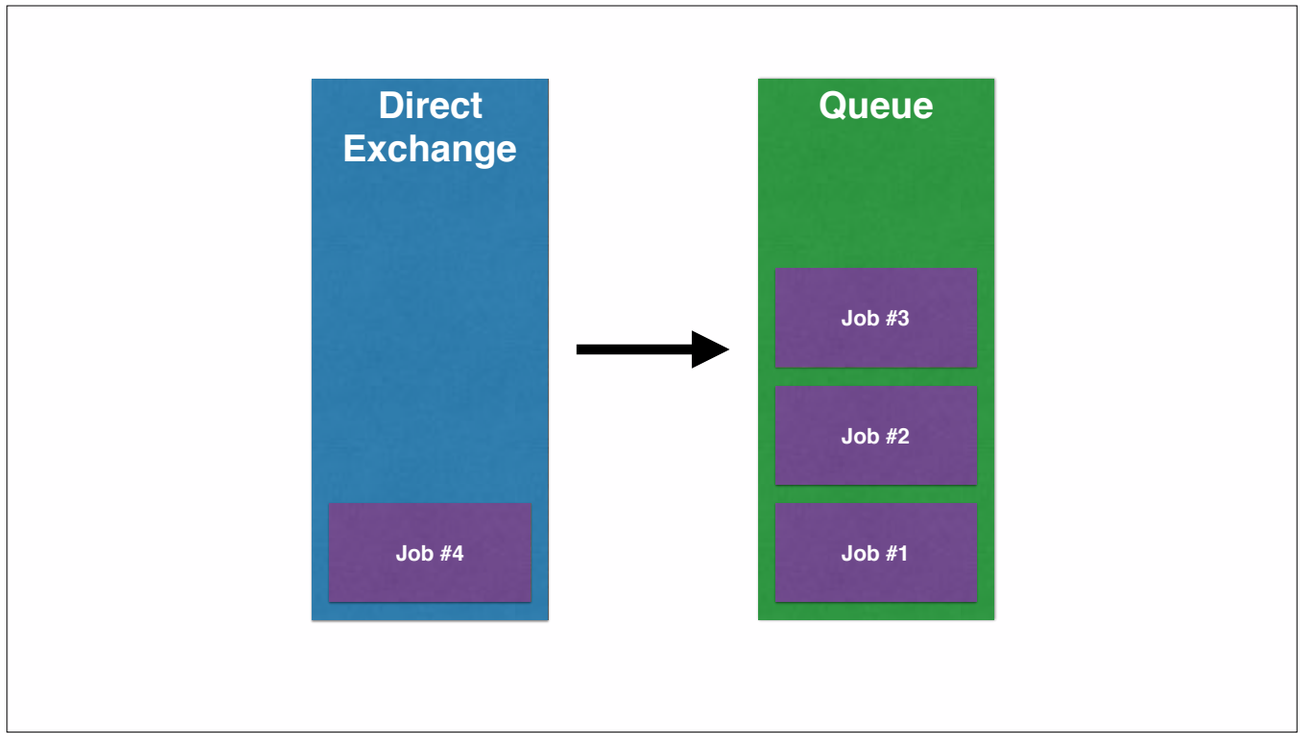
This brings us to exchanges. As they are the only Tab that most people won’t know what it does.



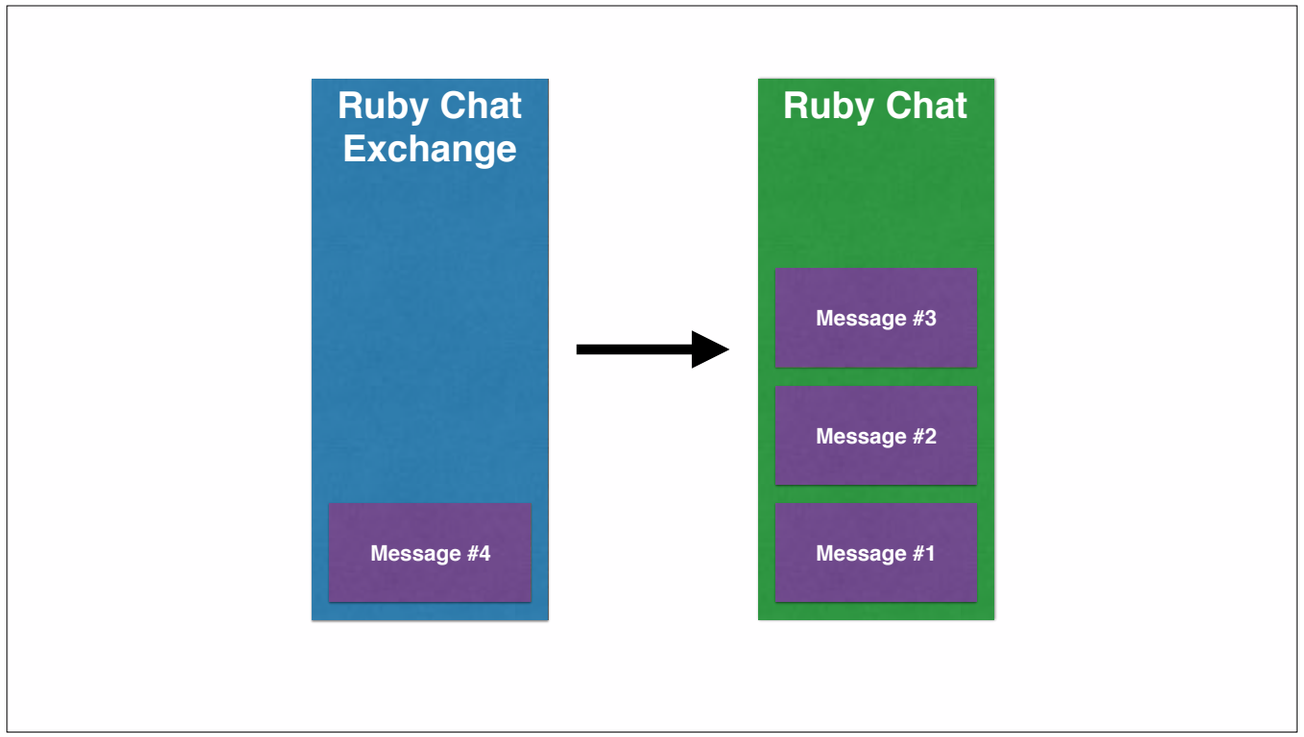
Exchanges are a feature of AMPQ. Using AMPQ it’s actually impossible to push something directly to a queue.



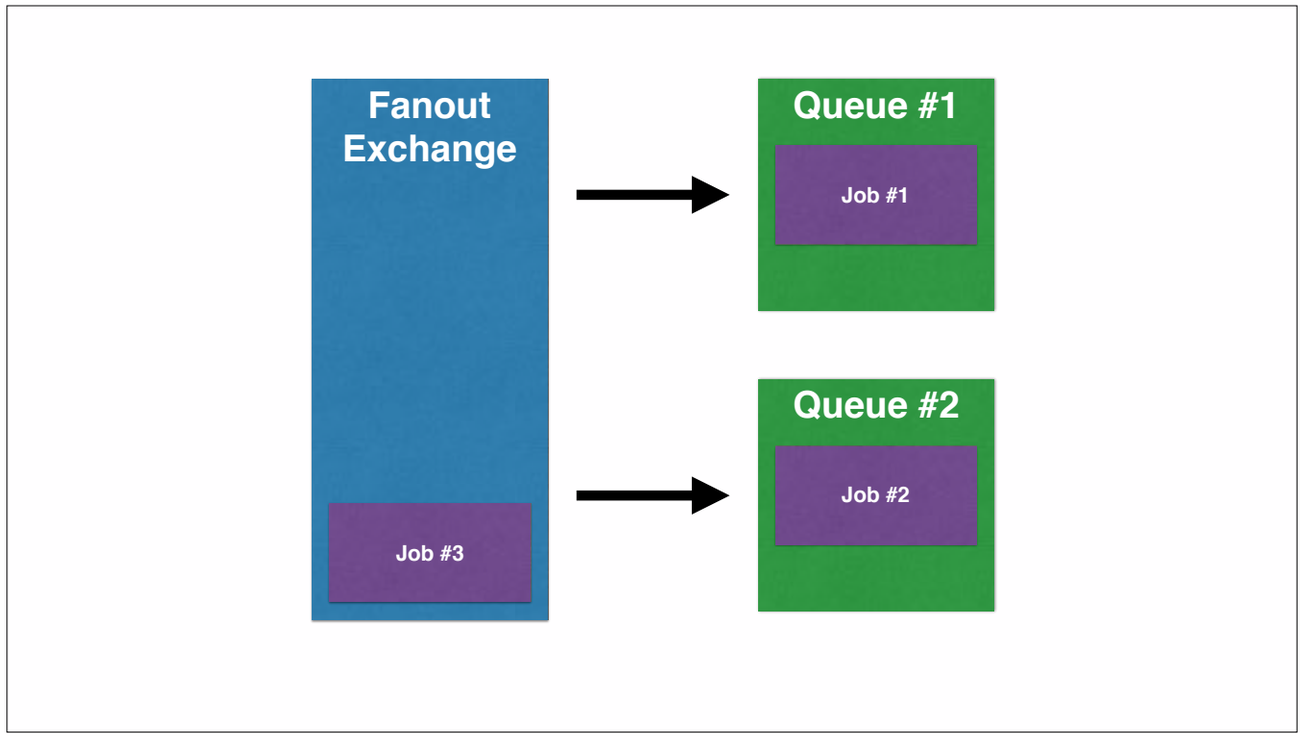
In AMPQ, it’s only possible to push a message to an exchange, and the exchange decides in which queues to put the message.



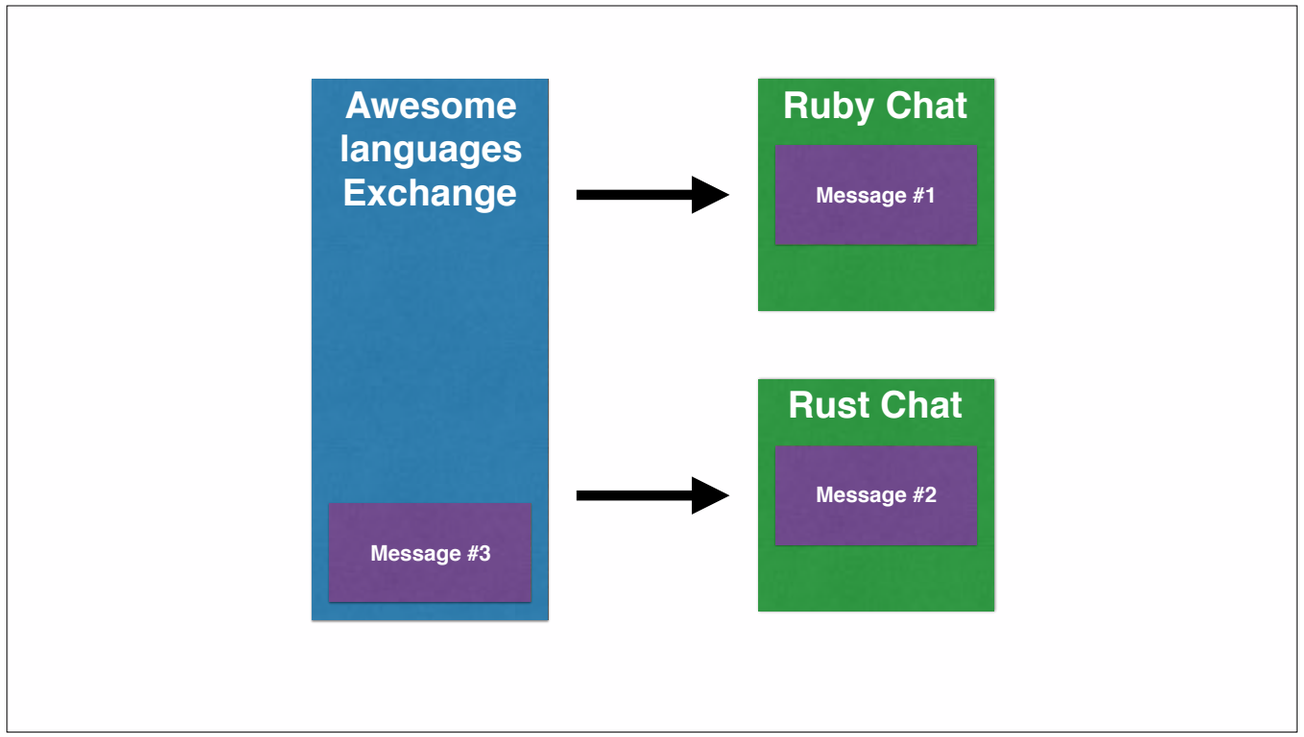
There are four types of exchanges. The above is a direct exchange. Messages passed to a direct exchange are directly passed to a single queue. This is the way that Sidekiq works. You push a job to a queue and it gets processed.



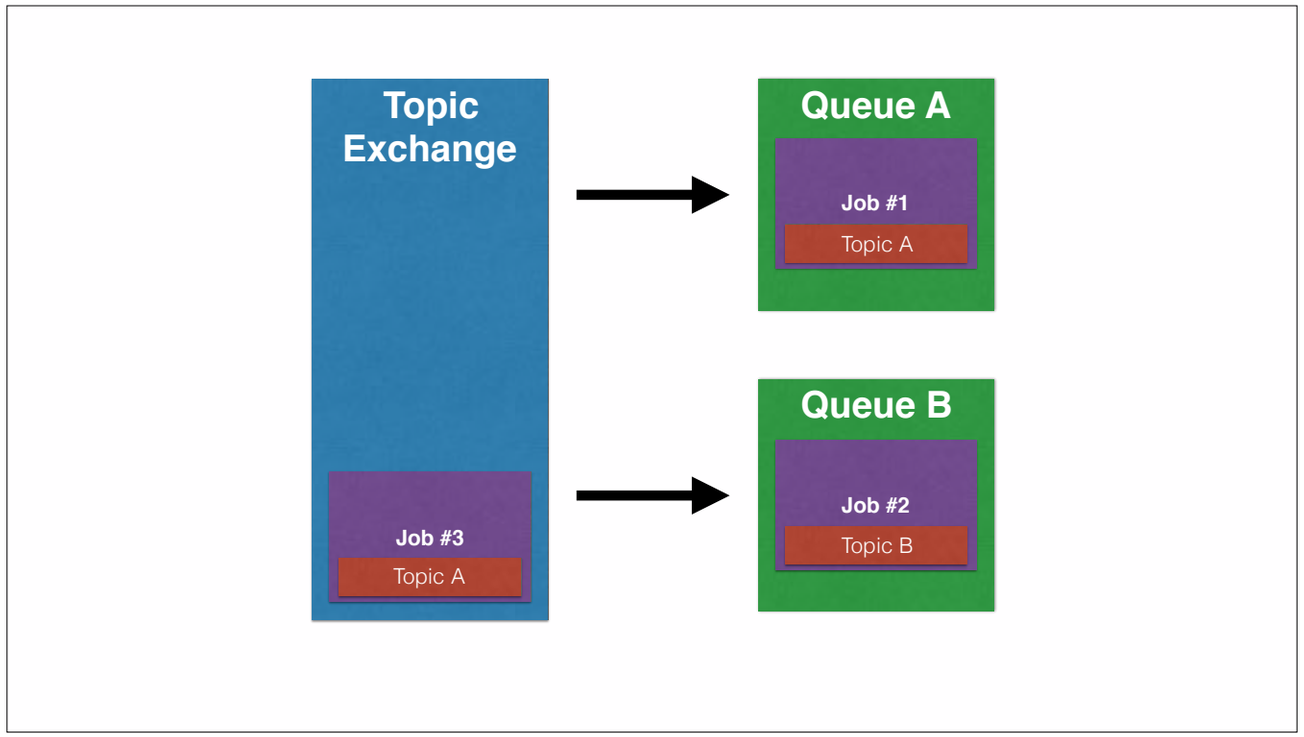
If we were building a chat, direct exchanges would be the same as sending a message to a single chatroom. In this case Message #4 would be delivery to the Ruby Chat, and only to the Ruby chat.



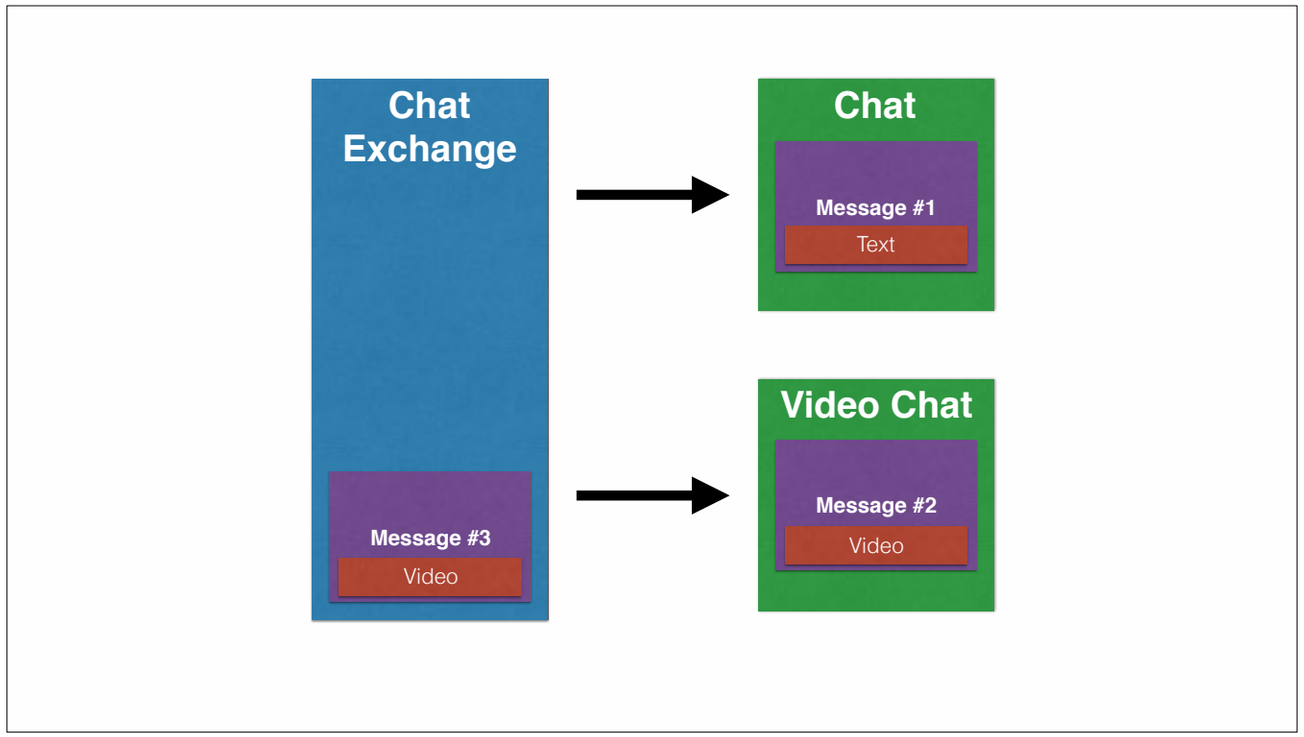
Then there are fanout exchanges. They send out the message to all bound queues.



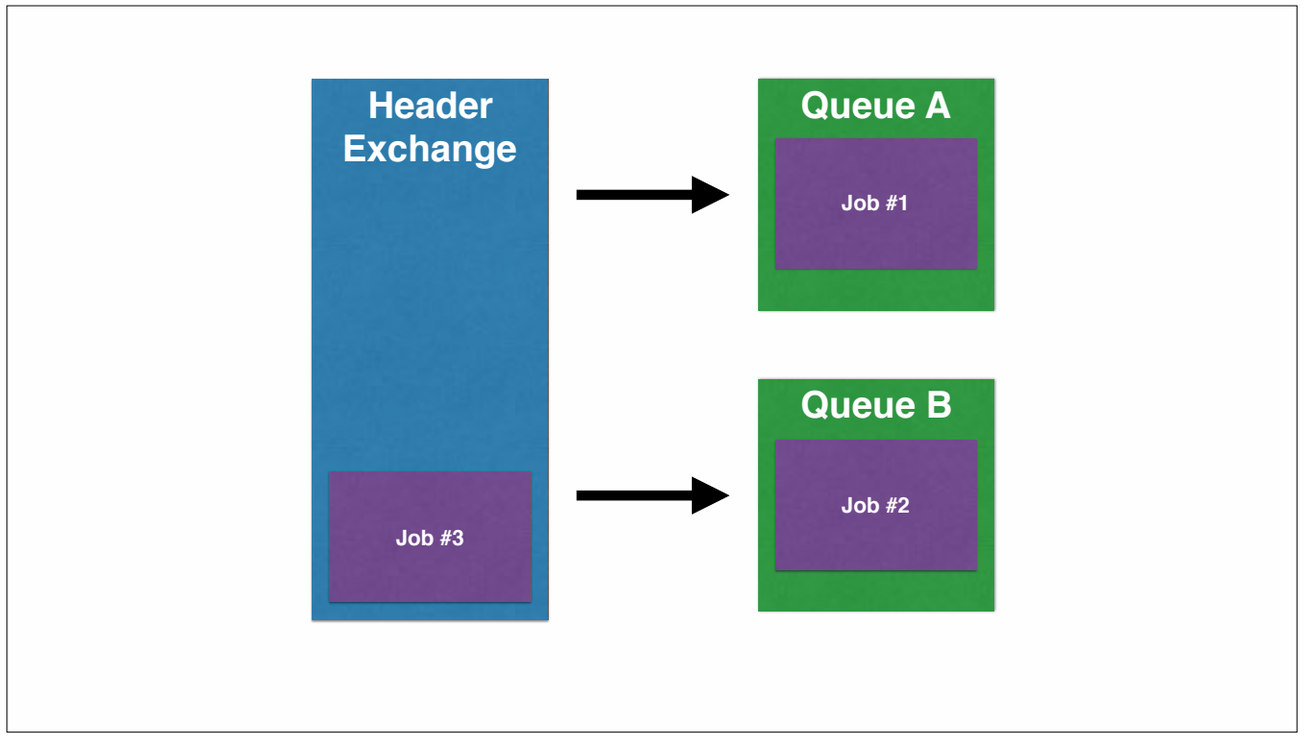
A practical example for this is a broadcast in a chat room. Let’s say we want to broadcast a message to all chatrooms of awesome languages. This is a good use cast for a fanout exchange. In the above case, Message #3 will be delivered both to the Ruby Chat queue and the Rust chat queue.



Topic exchanges are simple routers. With each message you publish you can specify a topic. The exchange will look at that topic and send the message to all queues bound to that topic.



In our chat example, you would use a topic exchange to direct messages to specific pipelines. E.g. if your users can upload images and video, and you want to process them before sending them in the chat. Video messages would get tagged and directed to a queue that processes them first, and then sends them out.



Lastly, there are header exchanges. They are the same as topic exchanges but more powerful, they can use nearly any part of the message to determine where to rout the message. The sample for this is the same as for topic exchanges, therefore I'll skip it.

But what would you use the?

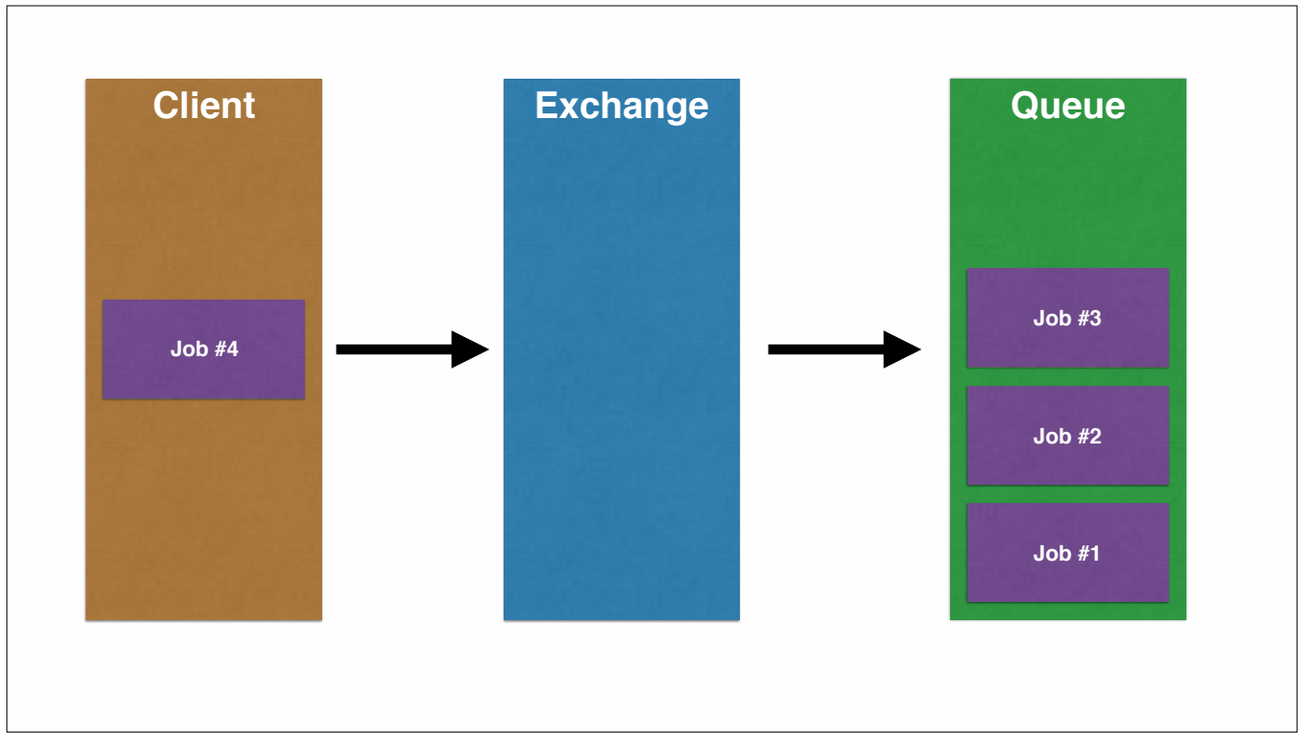
Why would you use exchanges if you can do all that logic in plain Ruby?
The first reason is speed. Doing this kind of routing is quite slow with Ruby.
The second exactly-once-delivery, if your Ruby process restarts it will get to send out all the messages again.
The third reason is a pragmatic one. That logic is not in your codebase. In other words, you don’t have to maintain it.
The final reason is deprivation. With exchanges it’s much easier to deprecate a service, simple point the exchange to a new service and you are done.

03 SPECIAL FEATURES

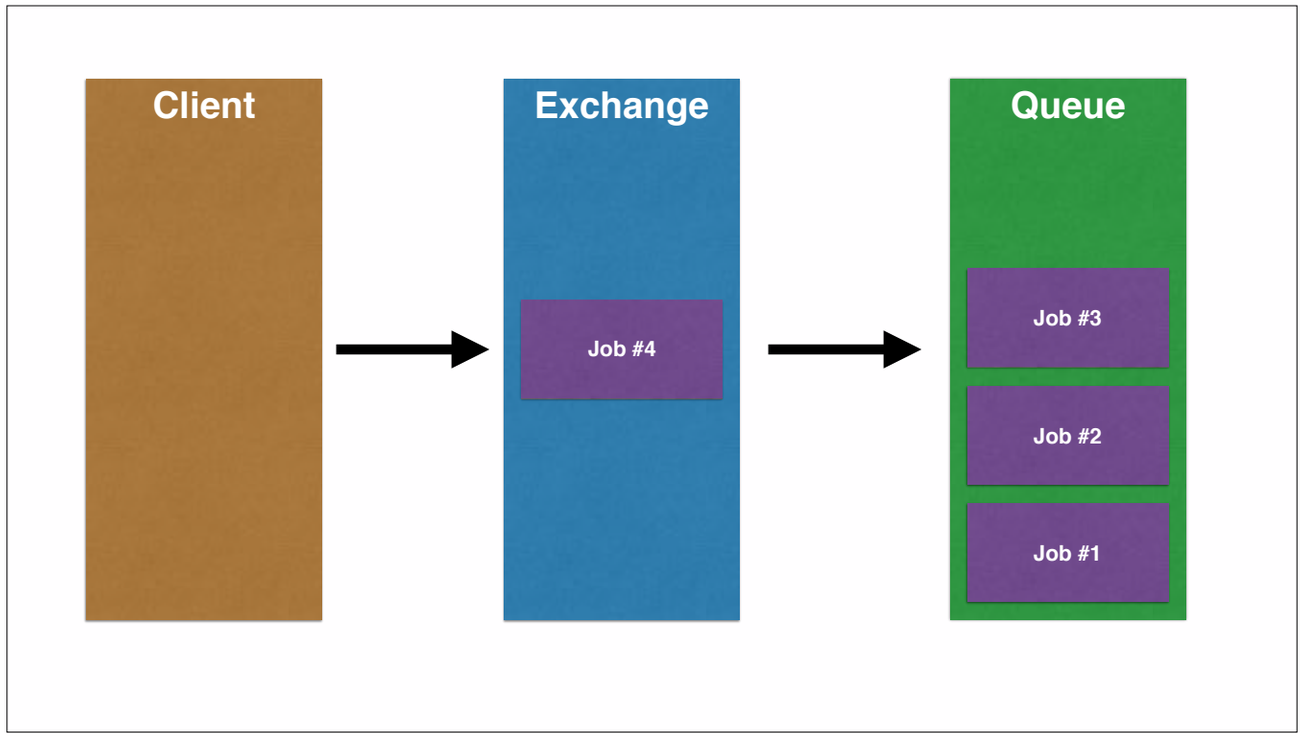
RabbitMQ comes with a couple special features non-standard for AMPQ

Direct reply-to

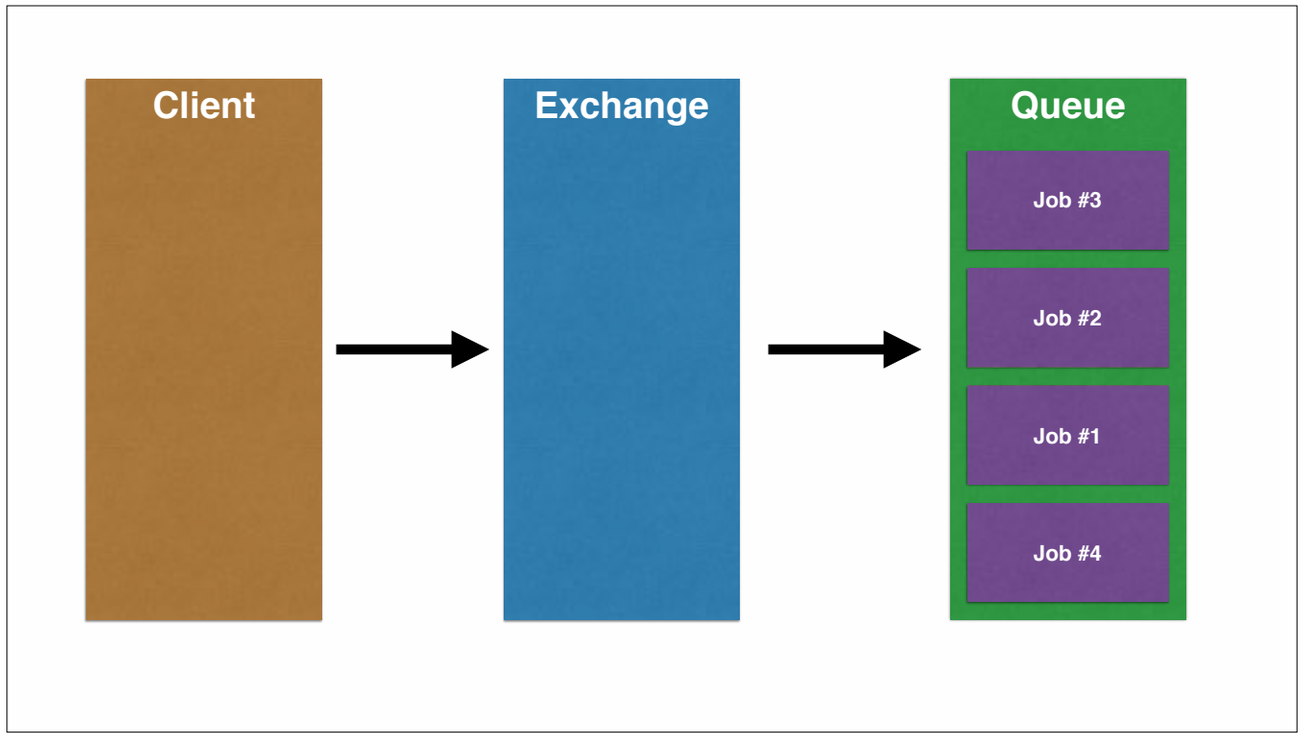
The first, and perhaps most useful feature is direct reply-to. It enables synchronous IPC.



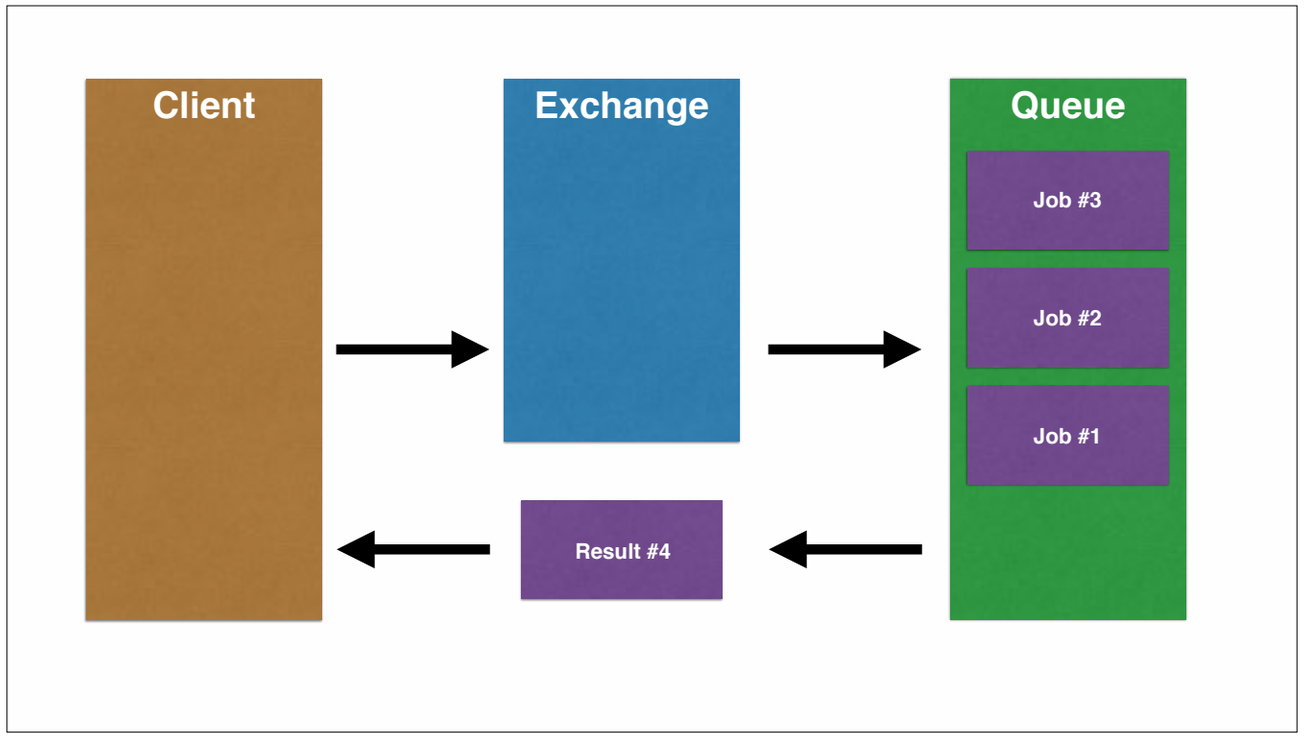
When a client sends a message



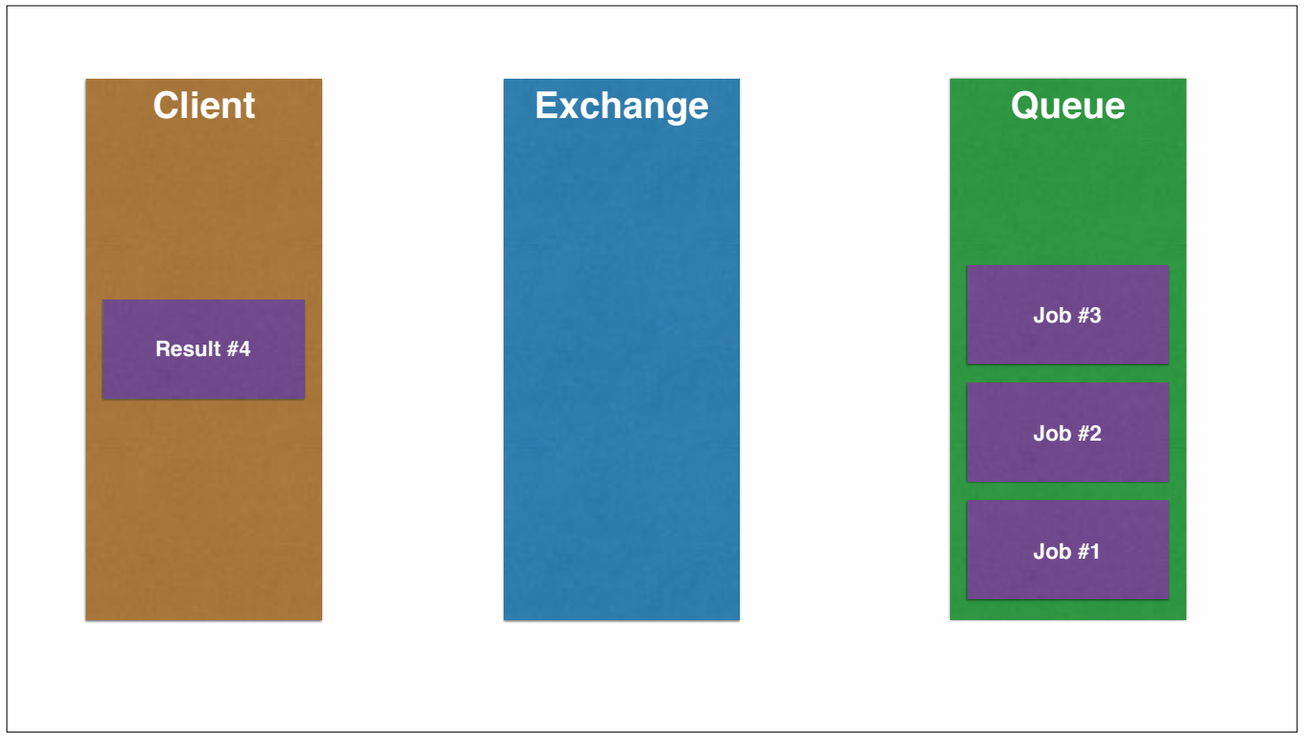
It’s received by the exchange and routed to a queue



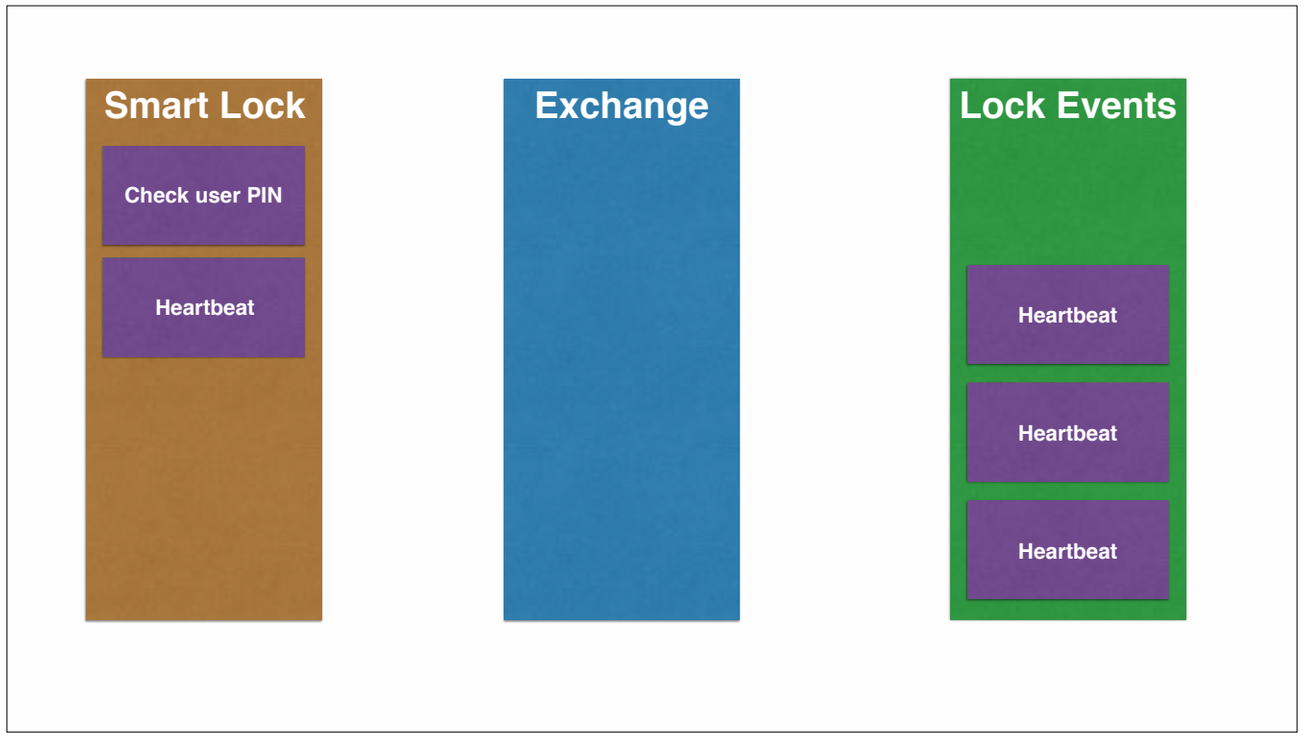
At the queue it’s handled like a priority message. e.g. it gets processed first.



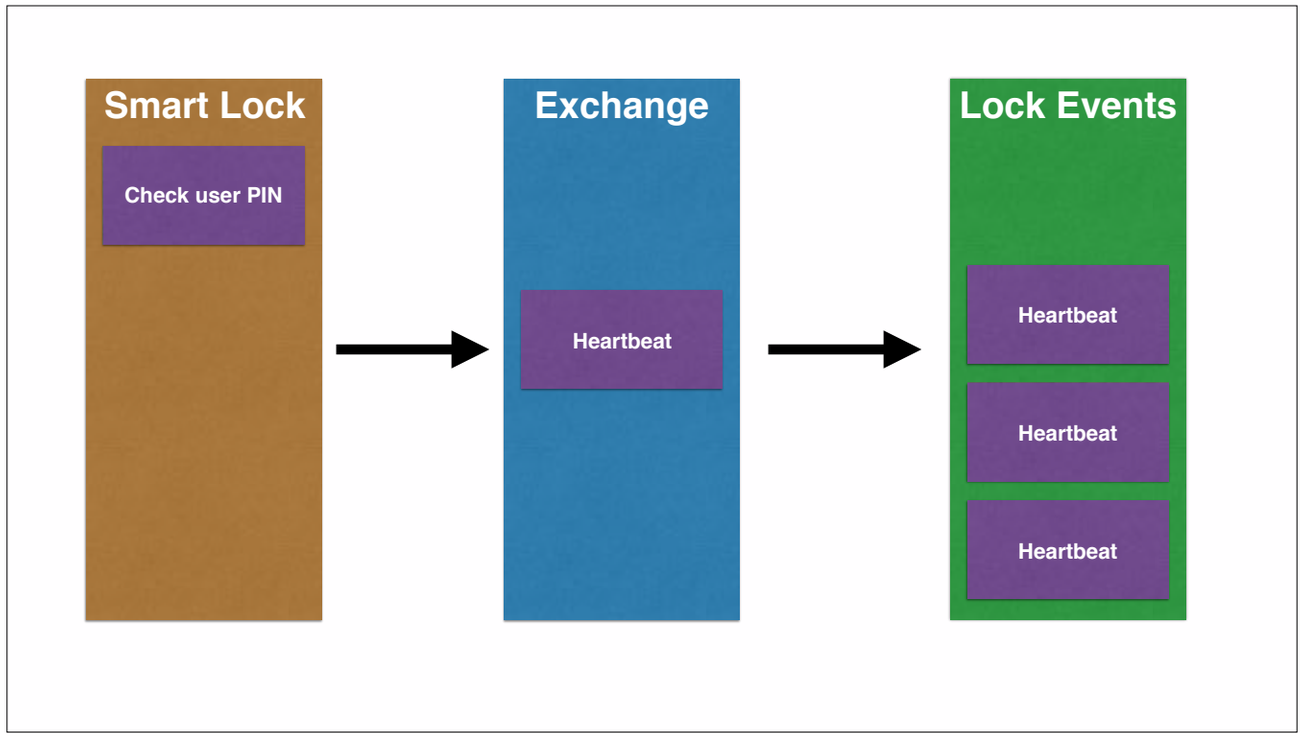
Now the special part. The result is passed back to the client! Directly!



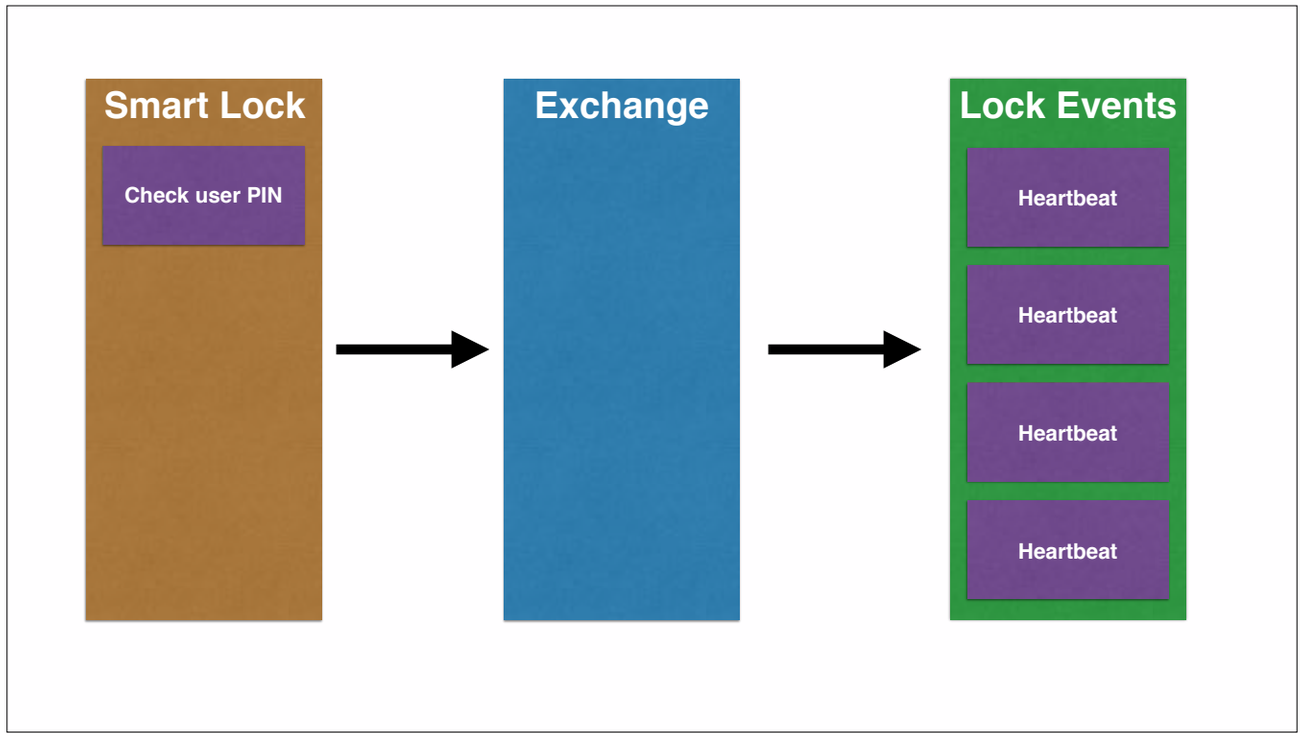
And then execution continues as normal.



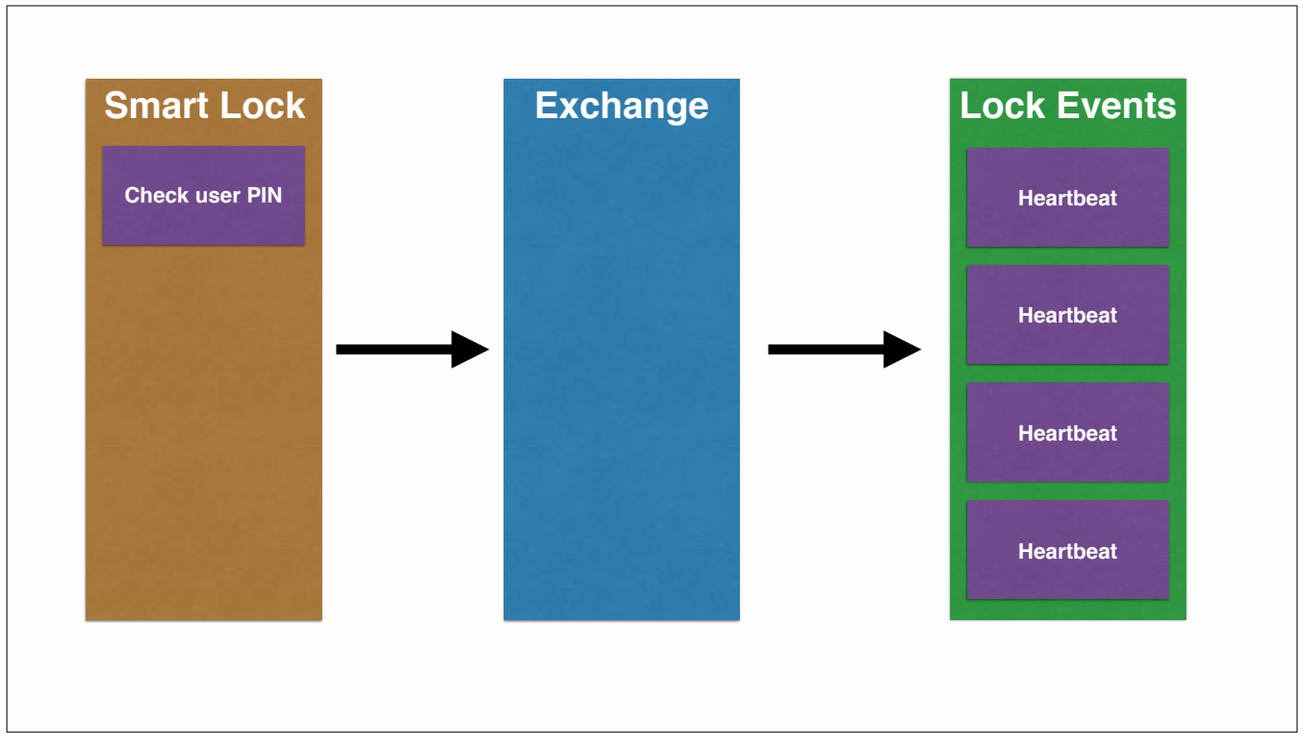
A real world example with a smart lock would be as follows. We have a smart lock, it needs to send out a regular heartbeat to indicate that ti’s still online and connected. But it can also check if a user’s pin is correct.



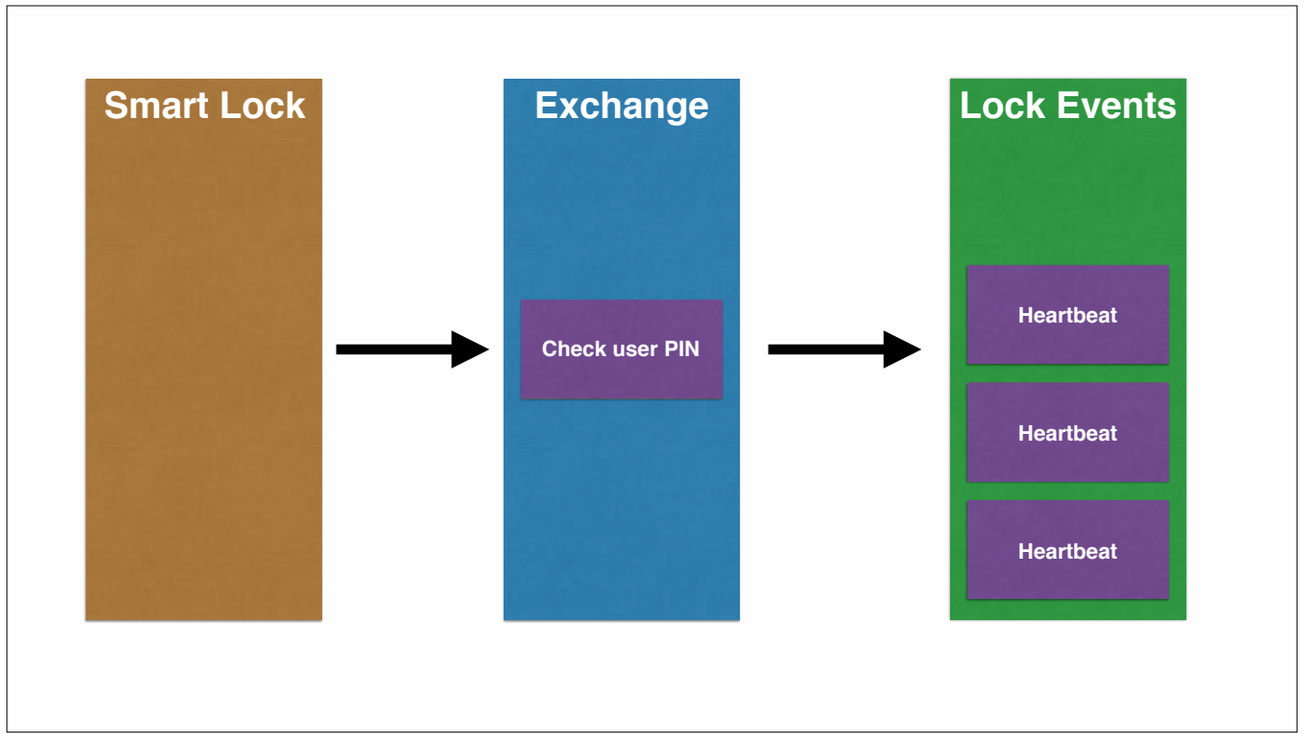
A heartbeat is not that important of a event that we need to stop the world and process it, so we put it into the queue



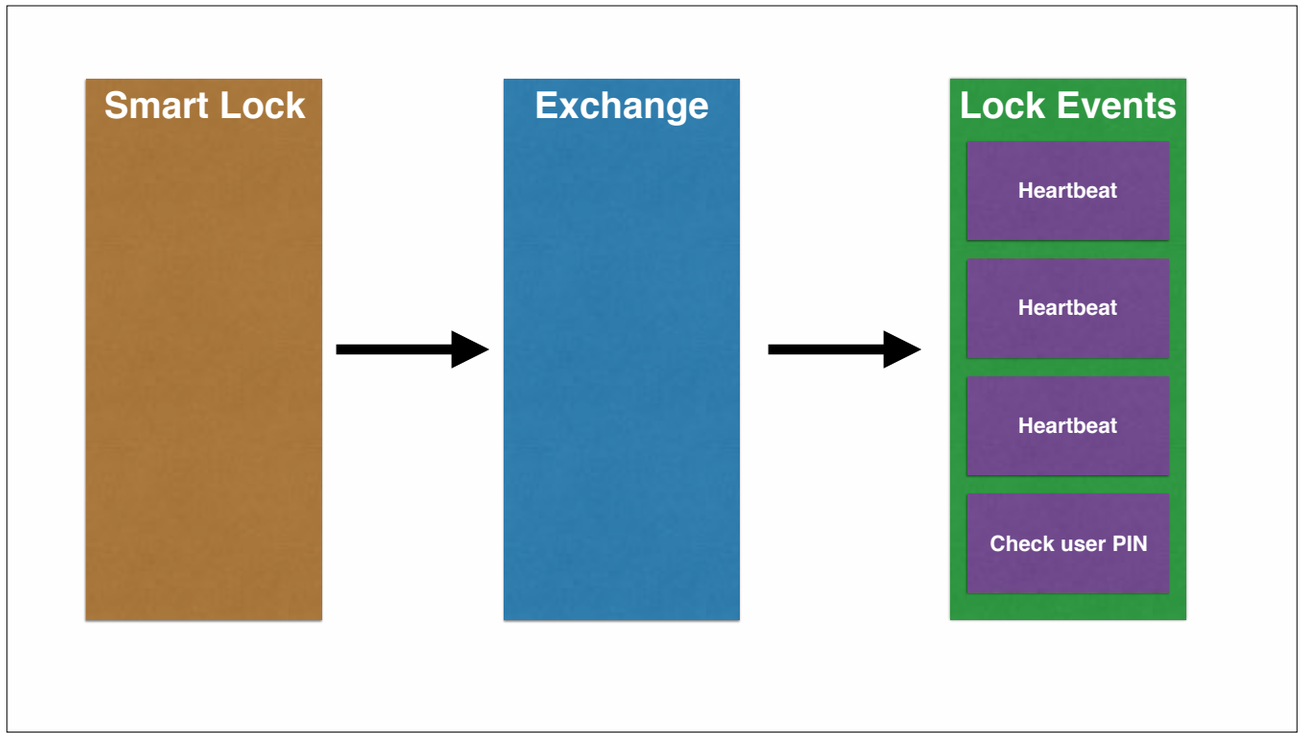
A heartbeat is not that important of a event that we need to stop the world and process it, so we put it into the queue



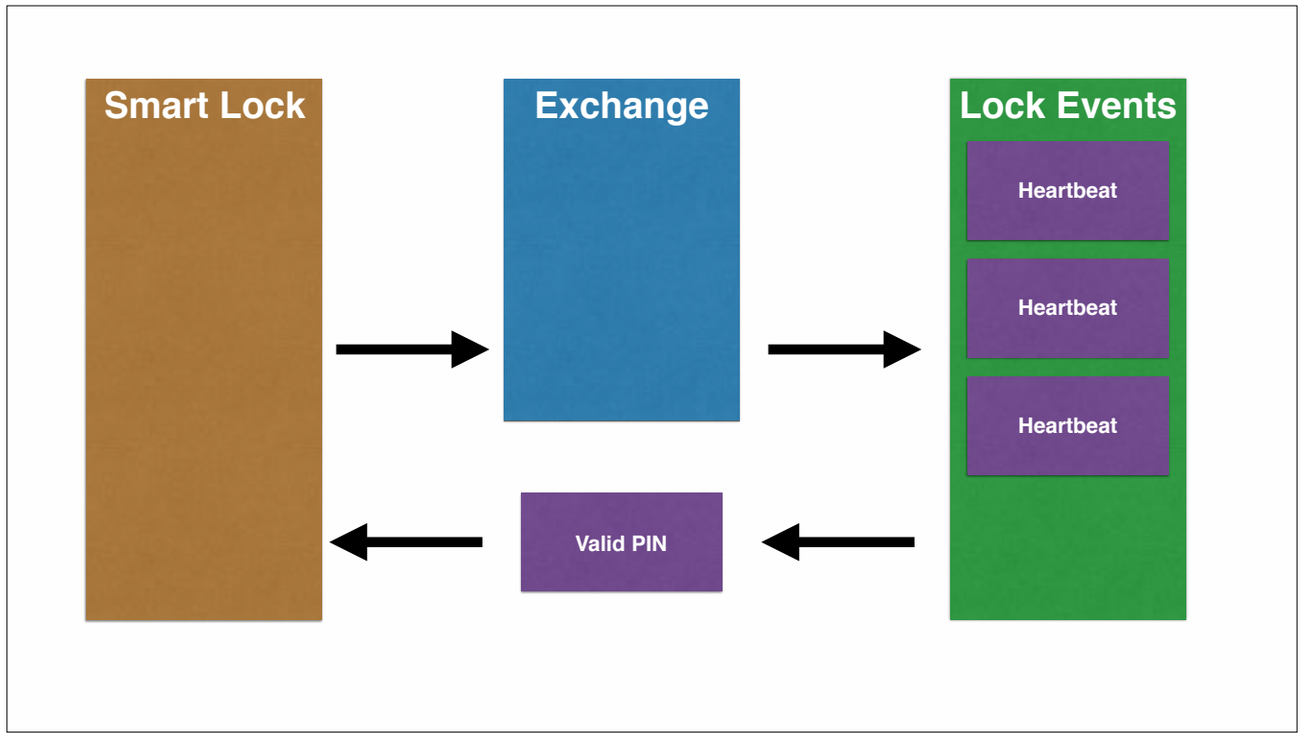
But a PIN check has to be as fast as possible. We don’t want to penalise the user for our server load.



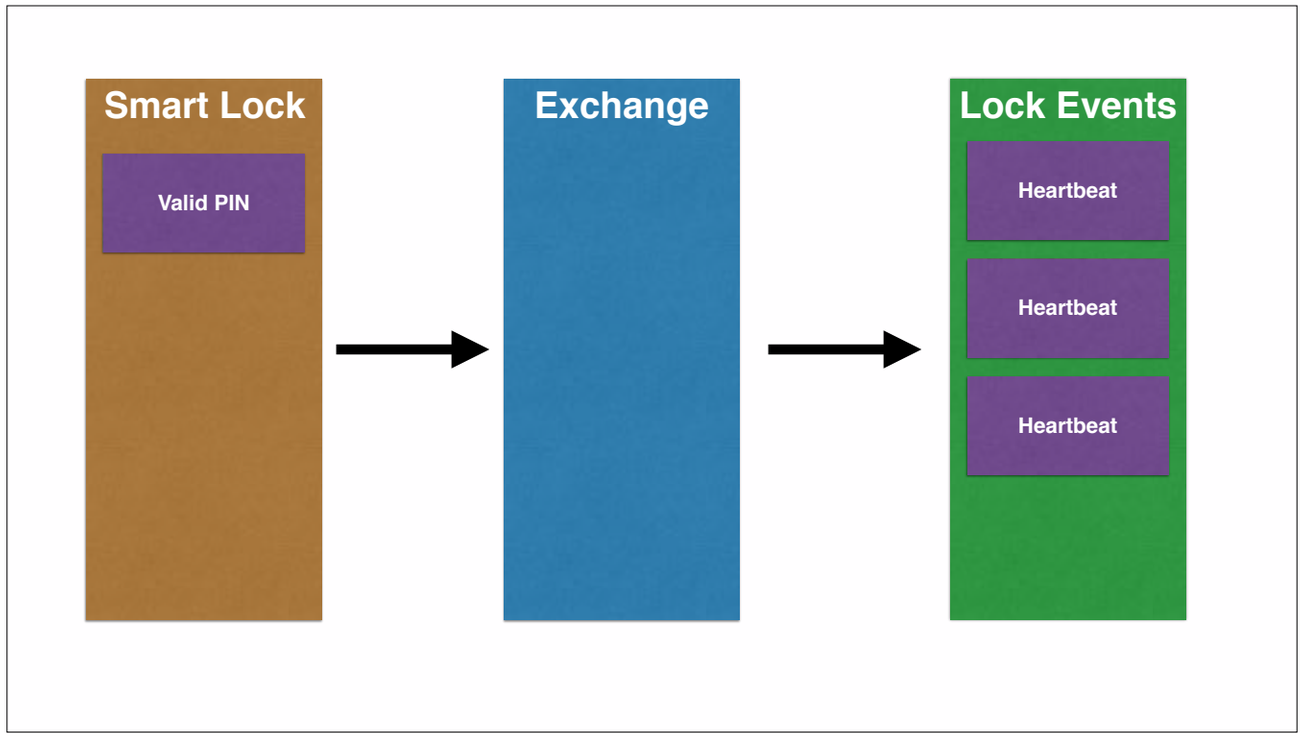
But a PIN check has to be as fast as possible. We don’t want to penalise the user for our server load.



But a PIN check has to be as fast as possible. We don’t want to penalise the user for our server load.



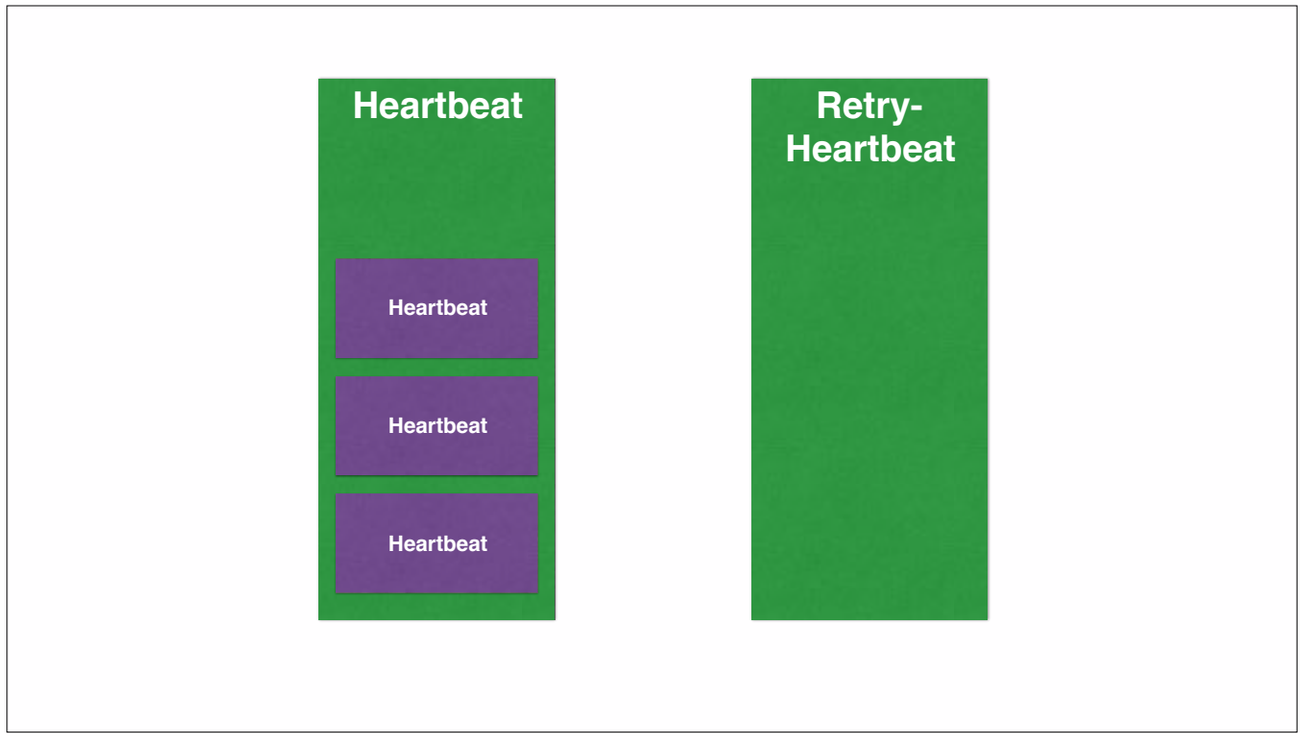
But a PIN check has to be as fast as possible. We don’t want to penalise the user for our server load.



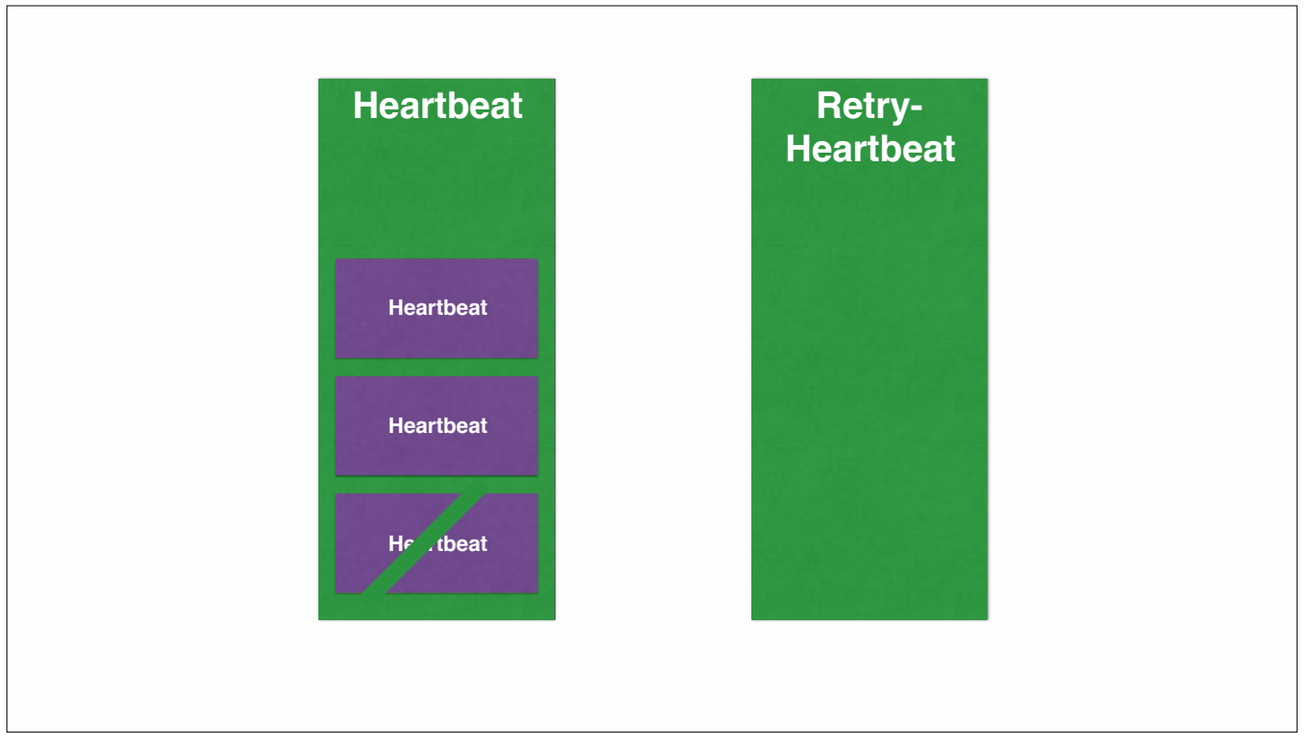
But a PIN check has to be as fast as possible. We don’t want to penalise the user for our server load.

Dead lettering

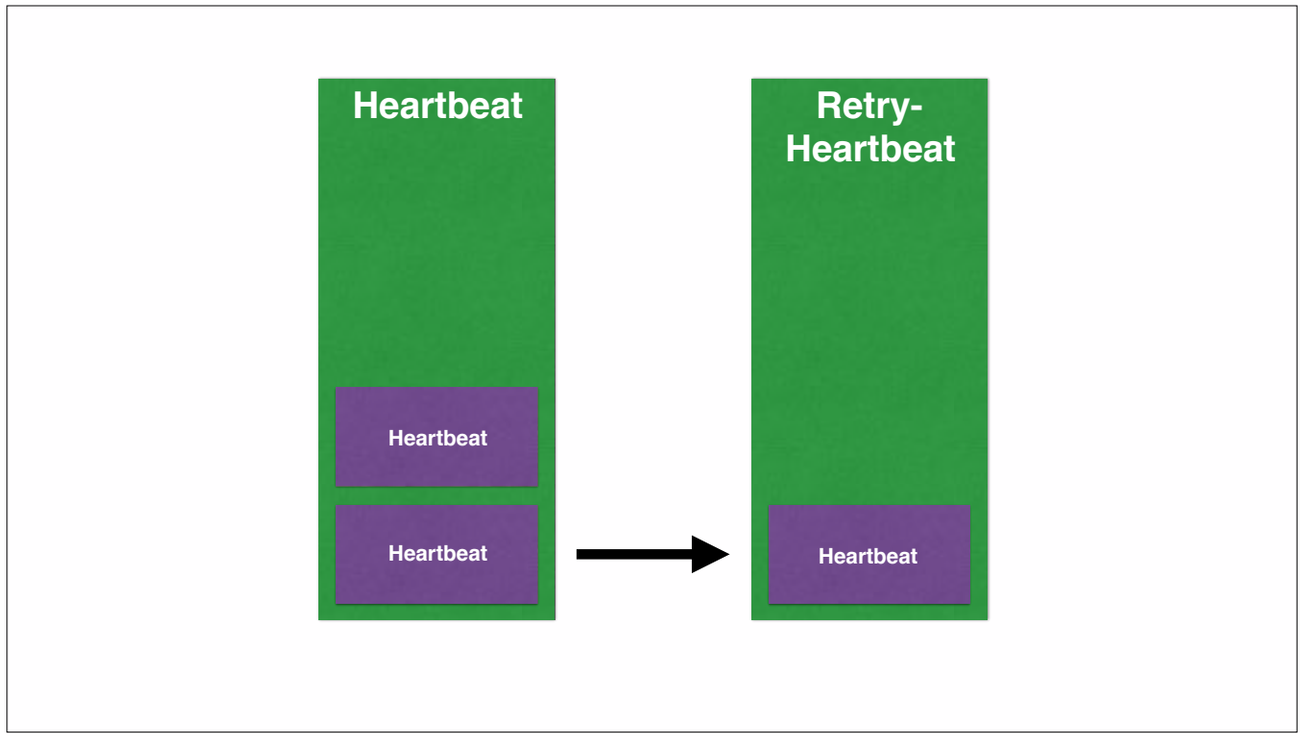
Dead lettering is a fault-tolerance feature for your application. If a message fails for any reason it will get sent to it’s corresponding dead-letter queue/exchange.



Now weh have two queues. One is our main queue. And the other is our retry queue.



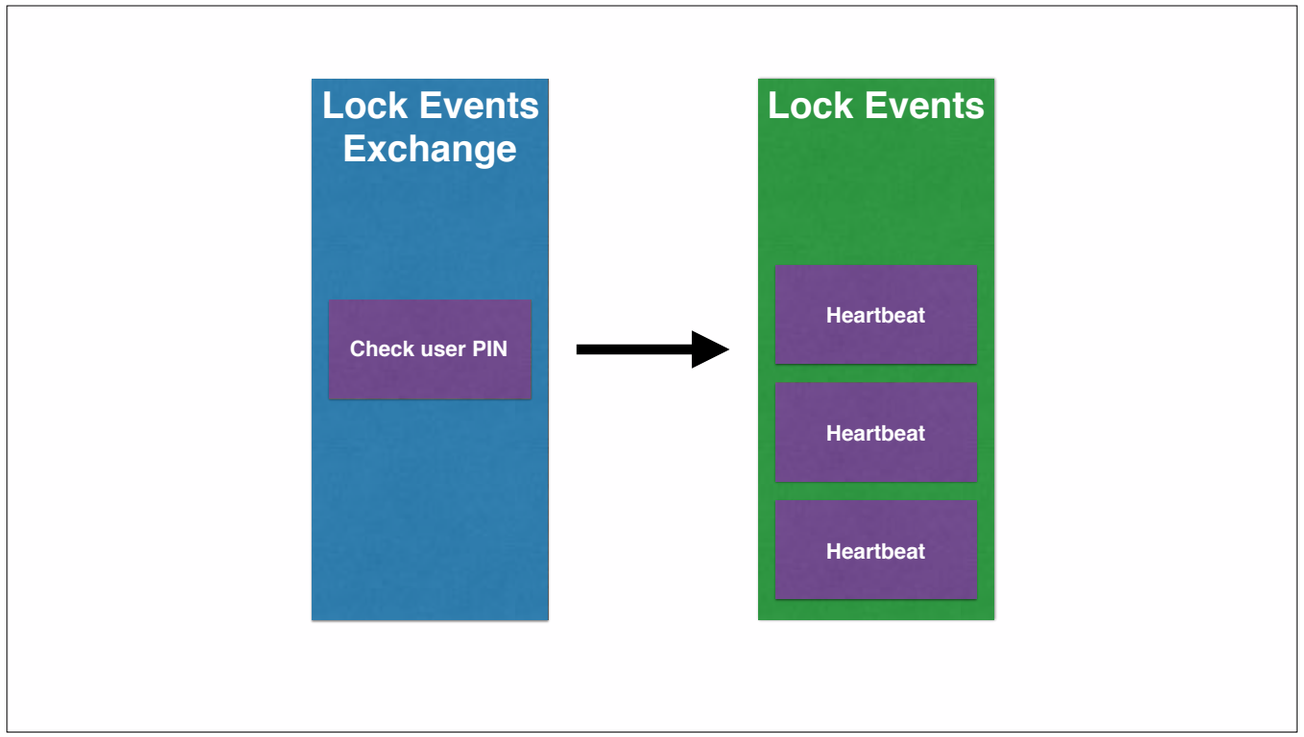
If a job in the main queue fails



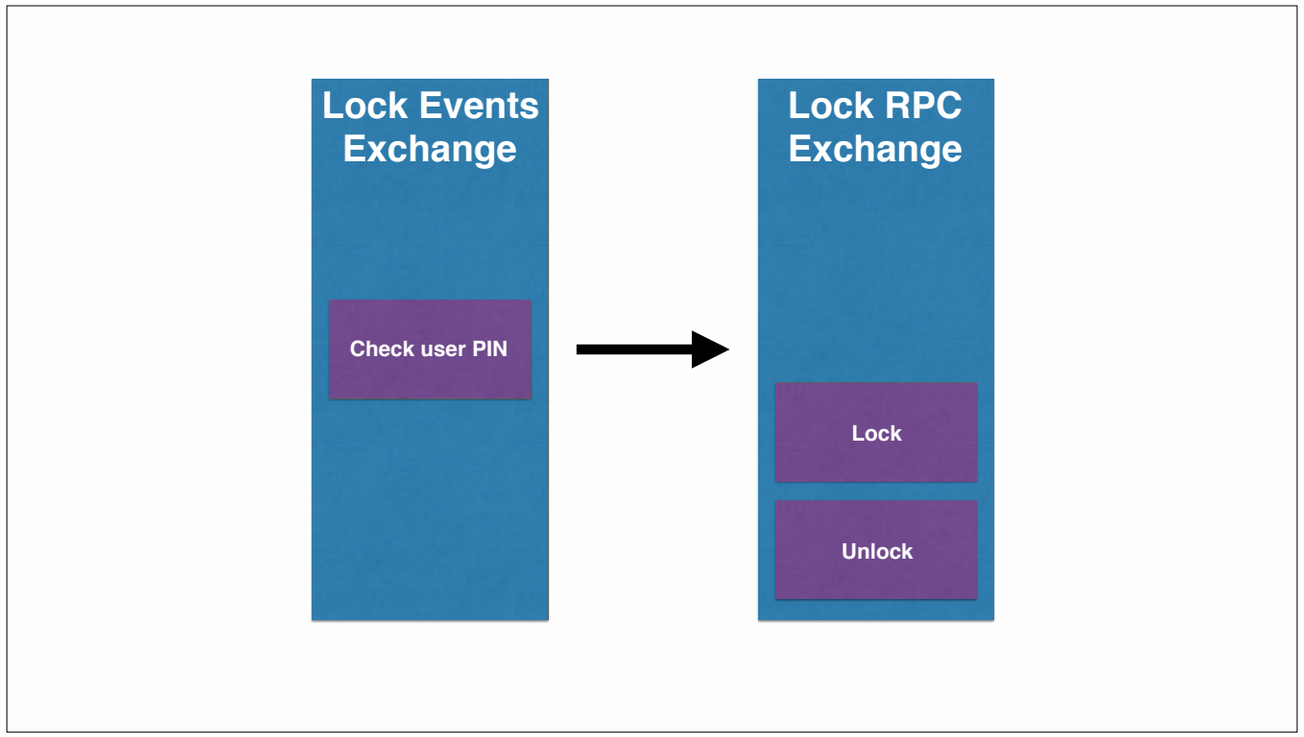
It’s transferred to the retry queue. Now we can apply custom logic to handle this failure.

Alternate exchanges

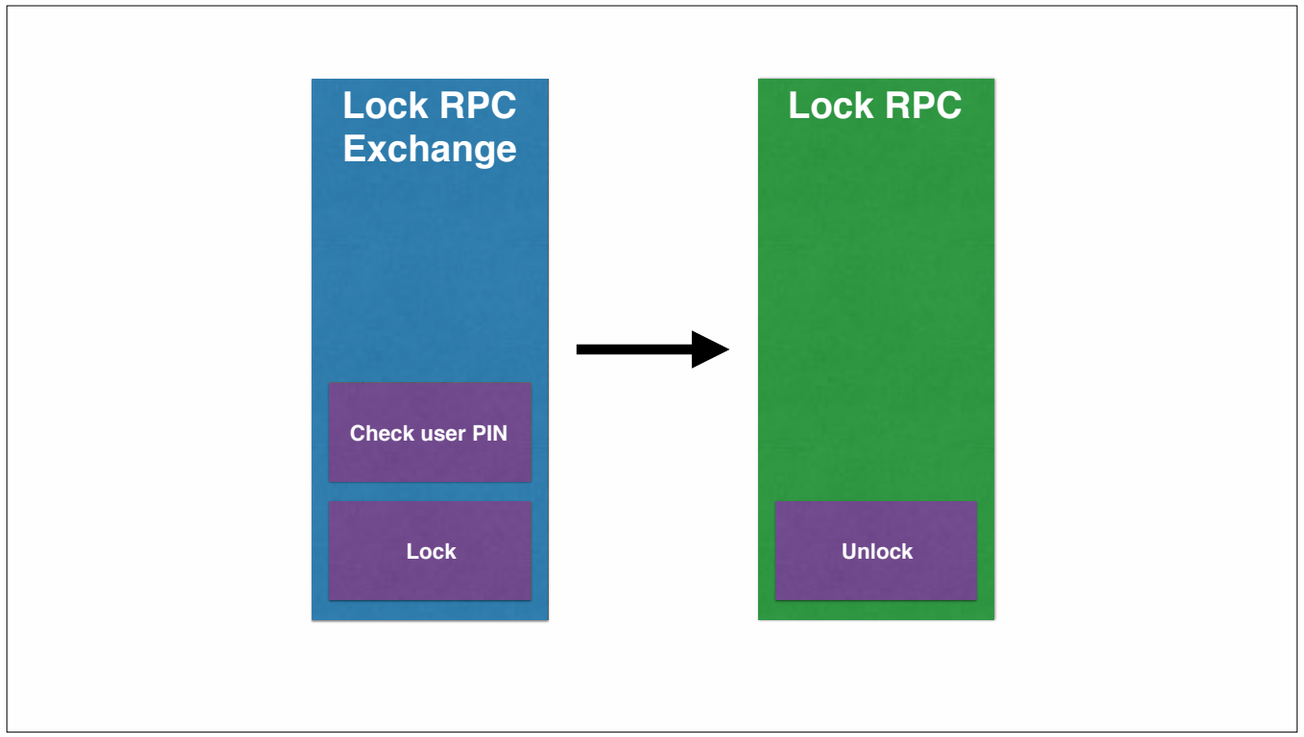
Allows you to specify an alternative exchange to send to if the primary exchange rejects the message



Let’s say that the lock events exchange its a topic exchange. If no topic corresponds to the message we sent.



Then the message is routed to the alternate exchange.



Then the alternate exchange handles the message as ususal. This feature is quite useful for deprecating services, and rerouting messages of deprecated or legacy services.

Priority consumers

Basically you can say that one consumer can tame more jobs than any other. This is useful if your app is running on a diverse set of machines. You can specify that higher capacity machines should get more jobs.

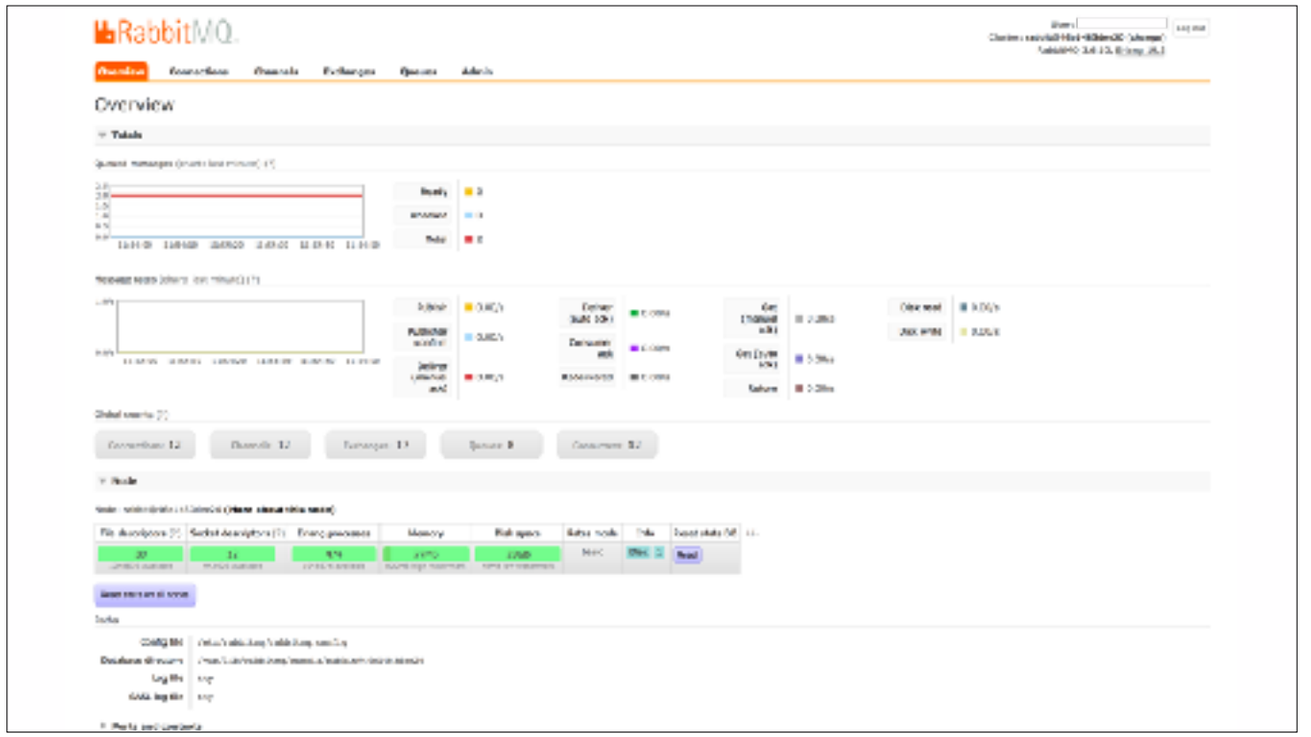
Priority queues

Basically queues where messages with higher priority get processed sooner.

And TTLs which specify an expirey time for the message.

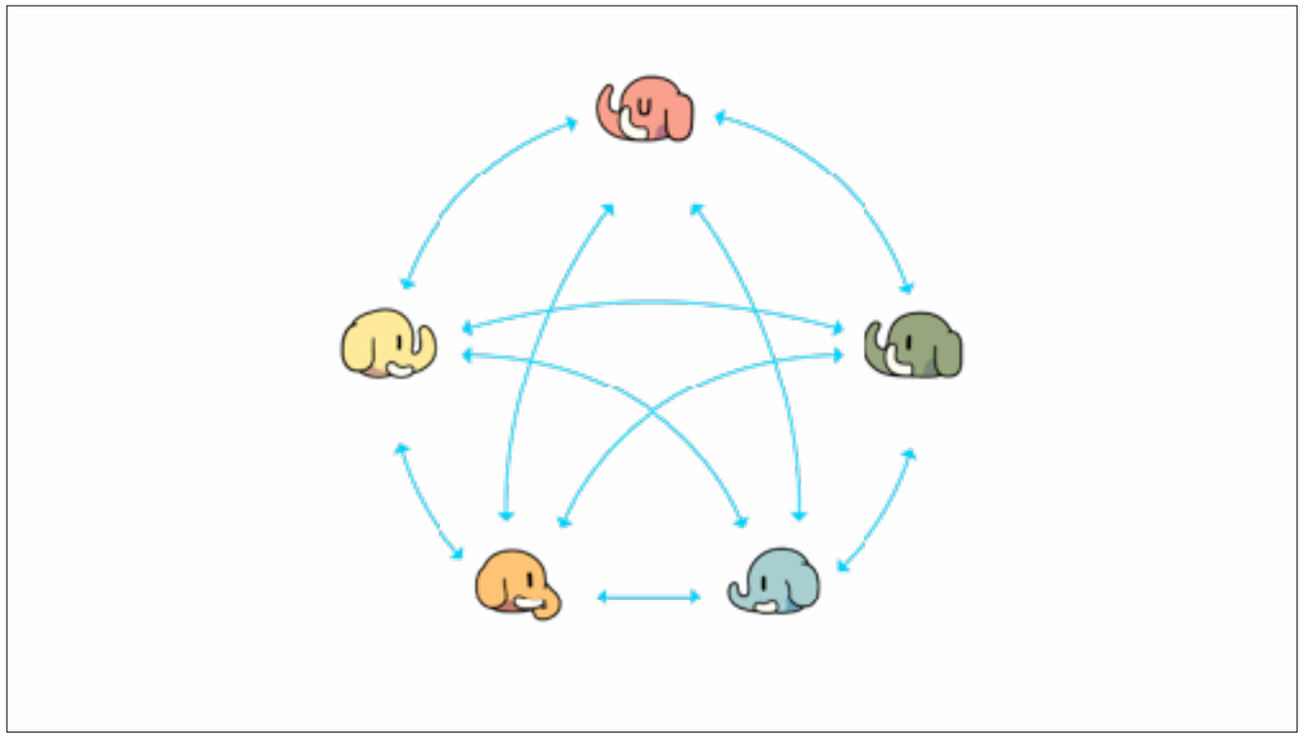
04 PLUGINS

Management


RabbitMQ
Overview

MQTT, STOMP and WebSockets

Federation



05

CONCLUSION

Debugging

Load balancing / Scaling

Memory consumption

Should I use it on my next project?

Monolith?
Probably not…

Except if you need assurances that your jobs will run.

Services?
YES!

Thank you for your patience. If anybody has questions I'll accept them now.
If you are shy, you can also contact me via Twitter or Email.