
Solid Queue

Solid Queue is a DB-based queuing backend for Active Job, designed with simplicity and performance in mind.

Besides regular job enqueueing and processing, Solid Queue supports delayed jobs, concurrency controls, pausing queues, numeric priorities per job, priorities by queue order, and bulk enqueueing (`enqueue_all` for Active Job's `perform_all_later`). *Improvements to logging and instrumentation, a better CLI tool, a way to run within an existing process in “async” mode, and some way of specifying unique jobs are coming very soon.*

Solid Queue can be used with SQL databases such as MySQL, PostgreSQL or SQLite, and it leverages the `FOR UPDATE SKIP LOCKED` clause, if available, to avoid blocking and waiting on locks when polling jobs. It relies on Active Job for retries, discarding, error handling, serialization, or delays, and it's compatible with Ruby on Rails multi-threading.

Installation and usage

Add this line to your application's Gemfile:

```
1 gem "solid_queue"
```

And then execute:

```
1 $ bundle
```

Or install it yourself as:

```
1 $ gem install solid_queue
```

Now, you need to install the necessary migrations and configure the Active Job's adapter. You can do both at once using the provided generator:

```
1 $ bin/rails generate solid_queue:install
```

This will set `solid_queue` as the Active Job's adapter in production, and will copy the required migration over to your app.

Alternatively, you can add only the migration to your app:

```
1 $ bin/rails solid_queue:install:migrations
```

And set Solid Queue as your Active Job's queue backend manually, in your environment config:

```
1 # config/environments/production.rb
```

```
2 config.active_job.queue_adapter = :solid_queue
```

Alternatively, you can set only specific jobs to use Solid Queue as their backend if you're migrating from another adapter and want to move jobs progressively:

```
1 # app/jobs/my_job.rb
2
3 class MyJob < ApplicationJob
4   self.queue_adapter = :solid_queue
5   # ...
6 end
```

Finally, you need to run the migrations:

```
1 $ bin/rails db:migrate
```

After this, you'll be ready to enqueue jobs using Solid Queue, but you need to start Solid Queue's supervisor to run them.

```
1 $ bundle exec rake solid_queue:start
```

This will start processing jobs in all queues using the default configuration. See below to learn more about configuring Solid Queue.

For small projects, you can run Solid Queue on the same machine as your webserver. When you're ready to scale, Solid Queue supports horizontal scaling out-of-the-box. You can run Solid Queue on a separate server from your webserver, or even run `bundle exec rake solid_queue:start` on multiple machines at the same time. If you'd like to designate some machines to be only dispatchers or only workers, use `bundle exec rake solid_queue:dispatch` or `bundle exec rake solid_queue:work`, respectively.

Requirements

Besides Rails 7.1, Solid Queue works best with MySQL 8+ or PostgreSQL 9.5+, as they support `FOR UPDATE SKIP LOCKED`. You can use it with older versions, but in that case, you might run into lock waits if you run multiple workers for the same queue.

Configuration

Workers and dispatchers

We have three types of processes in Solid Queue: - *Workers* are in charge of picking jobs ready to run from queues and processing them. They work off the `solid_queue_ready_executions` table.

- *Dispatchers* are in charge of selecting jobs scheduled to run in the future that are due and *dispatching* them, which is simply moving them from the `solid_queue_scheduled_executions` table over to the `solid_queue_ready_executions` table so that workers can pick them up. They're also in charge of managing recurring tasks, dispatching jobs to process them according to their schedule. On top of that, they do some maintenance work related to concurrency controls. - The *supervisor* forks workers and dispatchers according to the configuration, controls their heartbeats, and sends them signals to stop and start them when needed.

By default, Solid Queue will try to find your configuration under `config/solid_queue.yml`, but you can set a different path using the environment variable `SOLID_QUEUE_CONFIG`. This is what this configuration looks like:

```
1 production:
2   dispatchers:
3     - polling_interval: 1
4       batch_size: 500
5       concurrency_maintenance_interval: 300
6   workers:
7     - queues: "*"
8       threads: 3
9       polling_interval: 2
10    - queues: [ real_time, background ]
11      threads: 5
12      polling_interval: 0.1
13      processes: 3
```

Everything is optional. If no configuration is provided, Solid Queue will run with one dispatcher and one worker with default settings.

- `polling_interval`: the time interval in seconds that workers and dispatchers will wait before checking for more jobs. This time defaults to 1 second for dispatchers and 0.1 seconds for workers.
- `batch_size`: the dispatcher will dispatch jobs in batches of this size. The default is 500.
- `concurrency_maintenance_interval`: the time interval in seconds that the dispatcher will wait before checking for blocked jobs that can be unblocked. Read more about concurrency controls to learn more about this setting. It defaults to 600 seconds.
- `queues`: the list of queues that workers will pick jobs from. You can use `*` to indicate all queues (which is also the default and the behaviour you'll get if you omit this). You can provide a single queue, or a list of queues as an array. Jobs will be polled from those queues in order, so for example, with `[real_time, background]`, no jobs will be taken from `background` unless there aren't any more jobs waiting in `real_time`. You can also provide a prefix with a wildcard to match queues starting with a prefix. For example:

```
1 staging:
2   workers:
3     - queues: staging*
4       threads: 3
5       polling_interval: 5
```

This will create a worker fetching jobs from all queues starting with `staging`. The wildcard `*` is only allowed on its own or at the end of a queue name; you can't specify queue names such as `*_some_queue`. These will be ignored.

Finally, you can combine prefixes with exact names, like `[staging*, background]`, and the behaviour with respect to order will be the same as with only exact names.

- `threads`: this is the max size of the thread pool that each worker will have to run jobs. Each worker will fetch this number of jobs from their queue(s), at most and will post them to the thread pool to be run. By default, this is 3. Only workers have this setting.
- `processes`: this is the number of worker processes that will be forked by the supervisor with the settings given. By default, this is 1, just a single process. This setting is useful if you want to dedicate more than one CPU core to a queue or queues with the same configuration. Only workers have this setting.
- `concurrency_maintenance`: whether the dispatcher will perform the concurrency maintenance work. This is `true` by default, and it's useful if you don't use any concurrency controls and want to disable it or if you run multiple dispatchers and want some of them to just dispatch jobs without doing anything else.
- `recurring_tasks`: a list of recurring tasks the dispatcher will manage. Read more details about this one in the Recurring tasks section.

Queue order and priorities

As mentioned above, if you specify a list of queues for a worker, these will be polled in the order given, such as for the list `real_time, background`, no jobs will be taken from `background` unless there aren't any more jobs waiting in `real_time`.

Active Job also supports positive integer priorities when enqueueing jobs. In Solid Queue, the smaller the value, the higher the priority. The default is 0.

This is useful when you run jobs with different importance or urgency in the same queue. Within the same queue, jobs will be picked in order of priority, but in a list of queues, the queue order takes precedence, so in the previous example with `real_time, background`, jobs in the `real_time`

queue will be picked before jobs in the `background` queue, even if those in the `background` queue have a higher priority (smaller value) set.

We recommend not mixing queue order with priorities but either choosing one or the other, as that will make job execution order more straightforward for you.

Threads, processes and signals

Workers in Solid Queue use a thread pool to run work in multiple threads, configurable via the `threads` parameter above. Besides this, parallelism can be achieved via multiple processes on one machine (configurable via different workers or the `processes` parameter above) or by horizontal scaling.

The supervisor is in charge of managing these processes, and it responds to the following signals:

- `TERM`, `INT`: starts graceful termination. The supervisor will send a `TERM` signal to its supervised processes, and it'll wait up to `SolidQueue.shutdown_timeout` time until they're done. If any supervised processes are still around by then, it'll send a `QUIT` signal to them to indicate they must exit.
- `QUIT`: starts immediate termination. The supervisor will send a `QUIT` signal to its supervised processes, causing them to exit immediately.

When receiving a `QUIT` signal, if workers still have jobs in-flight, these will be returned to the queue when the processes are deregistered.

If processes have no chance of cleaning up before exiting (e.g. if someone pulls a cable somewhere), in-flight jobs might remain claimed by the processes executing them. Processes send heartbeats, and the supervisor checks and prunes processes with expired heartbeats, which will release any claimed jobs back to their queues. You can configure both the frequency of heartbeats and the threshold to consider a process dead. See the section below for this.

Other configuration settings

Note: The settings in this section should be set in your `config/application.rb` or your environment config like this: `config.solid_queue.silence_polling = true`

There are several settings that control how Solid Queue works that you can set as well: - `logger`: the logger you want Solid Queue to use. Defaults to the app logger. - `app_executor`: the Rails executor used to wrap asynchronous operations, defaults to the app executor - `on_thread_error`: custom lambda/Proc to call when there's an error within a thread that takes the exception raised as argument. Defaults to

```
1 -> (exception) { Rails.error.report(exception, handled: false) }
```

-
- `connects_to`: a custom database configuration that will be used in the abstract `SolidQueue::Record` Active Record model. This is required to use a different database than the main app. For example:

```
1 # Use a separate DB for Solid Queue
2 config.solid_queue.connects_to = { database: { writing: :
    solid_queue_primary, reading: :solid_queue_replica } }
```

- `use_skip_locked`: whether to use `FOR UPDATE SKIP LOCKED` when performing locking reads. This will be automatically detected in the future, and for now, you'd only need to set this to **false** if your database doesn't support it. For MySQL, that'd be versions < 8, and for PostgreSQL, versions < 9.5. If you use SQLite, this has no effect, as writes are sequential.
- `process_heartbeat_interval`: the heartbeat interval that all processes will follow—defaults to 60 seconds.
- `process_alive_threshold`: how long to wait until a process is considered dead after its last heartbeat—defaults to 5 minutes.
- `shutdown_timeout`: time the supervisor will wait since it sent the `TERM` signal to its supervised processes before sending a `QUIT` version to them requesting immediate termination—defaults to 5 seconds.
- `silence_polling`: whether to silence Active Record logs emitted when polling for both workers and dispatchers—defaults to **true**.
- `supervisor_pidfile`: path to a pidfile that the supervisor will create when booting to prevent running more than one supervisor in the same host, or in case you want to use it for a health check. It's `nil` by default.
- `preserve_finished_jobs`: whether to keep finished jobs in the `solid_queue_jobs` table—defaults to **true**.
- `clear_finished_jobs_after`: period to keep finished jobs around, in case `preserve_finished_jobs` is true—defaults to 1 day. **Note:** Right now, there's no automatic cleanup of finished jobs. You'd need to do this by periodically invoking `SolidQueue::Job.clear_finished_in_batches`, but this will happen automatically in the near future.
- `default_concurrency_control_period`: the value to be used as the default for the `duration` parameter in concurrency controls. It defaults to 3 minutes.
- `enqueue_after_transaction_commit`: whether the job queuing is deferred to after the current Active Record transaction is committed. The default is **false**. Read more.

Concurrency controls

Solid Queue extends Active Job with concurrency controls, that allows you to limit how many jobs of a certain type or with certain arguments can run at the same time. When limited in this way, jobs will be blocked from running, and they'll stay blocked until another job finishes and unblocks them, or after the set expiry time (concurrency limit's *duration*) elapses. Jobs are never discarded or lost, only blocked.

```
1 class MyJob < ApplicationJob
2   limits_concurrency to: max_concurrent_executions, key: ->(arg1, arg2,
   **) { ... }, duration:
   max_interval_to_guarantee_concurrency_limit, group:
   concurrency_group
3
4   # ...
```

- `key` is the only required parameter, and it can be a symbol, a string or a proc that receives the job arguments as parameters and will be used to identify the jobs that need to be limited together. If the proc returns an Active Record record, the key will be built from its class name and `id`.
- `to` is 1 by default, and `duration` is set to `SolidQueue.default_concurrency_control_period` by default, which itself defaults to 3 `minutes`, but that you can configure as well.
- `group` is used to control the concurrency of different job classes together. It defaults to the job class name.

When a job includes these controls, we'll ensure that, at most, the number of jobs (indicated as `to`) that yield the same `key` will be performed concurrently, and this guarantee will last for `duration` for each job enqueued. Note that there's no guarantee about *the order of execution*, only about jobs being performed at the same time (overlapping).

For example:

```
1 class DeliverAnnouncementToContactJob < ApplicationJob
2   limits_concurrency to: 2, key: ->(contact) { contact.account },
   duration: 5.minutes
3
4   def perform(contact)
5     # ...
```

Where `contact` and `account` are `ActiveRecord` records. In this case, we'll ensure that at most two jobs of the kind `DeliverAnnouncementToContactJob` for the same account will run concurrently. If, for any reason, one of those jobs takes longer than 5 minutes or doesn't release its concurrency lock within 5 minutes of acquiring it, a new job with the same key might gain the lock.

Let's see another example using `group`:

```
1 class Box::MovePostingsByContactToDesignatedBoxJob < ApplicationJob
2   limits_concurrency key: ->(contact) { contact }, duration: 15.minutes
3   , group: "ContactActions"
4
5   def perform(contact)
6     # ...
```

```
1 class Bundle::RebundlePostingsJob < ApplicationJob
2   limits_concurrency key: ->(bundle) { bundle.contact }, duration: 15.
3   minutes, group: "ContactActions"
4
5   def perform(bundle)
6     # ...
```

In this case, if we have a `Box::MovePostingsByContactToDesignatedBoxJob` job enqueued for a contact record with id 123 and another `Bundle::RebundlePostingsJob` job enqueued simultaneously for a bundle record that references contact 123, only one of them will be allowed to proceed. The other one will stay blocked until the first one finishes (or 15 minutes pass, whatever happens first).

Note that the `duration` setting depends indirectly on the value for `concurrency_maintenance_interval` that you set for your dispatcher(s), as that'd be the frequency with which blocked jobs are checked and unblocked. In general, you should set `duration` in a way that all your jobs would finish well under that duration and think of the concurrency maintenance task as a failsafe in case something goes wrong.

Finally, failed jobs that are automatically or manually retried work in the same way as new jobs that get enqueued: they get in the queue for gaining the lock, and whenever they get it, they'll be run. It doesn't matter if they had gained the lock already in the past.

Failed jobs and retries

Solid Queue doesn't include any automatic retry mechanism, it relies on Active Job for this. Jobs that fail will be kept in the system, and a *failed execution* (a record in the `solid_queue_failed_executions` table) will be created for these. The job will stay there until manually discarded or re-enqueued. You can do this in a console as:

```
1 failed_execution = SolidQueue::FailedExecution.find(...) # Find the
2   failed execution related to your job
3   failed_execution.error # inspect the error
4
5   failed_execution.retry # This will re-enqueue the job as if it was
6   enqueued for the first time
7   failed_execution.discard # This will delete the job from the system
```

However, we recommend taking a look at `mission_control-jobs`, a dashboard where, among other things, you can examine and retry/discard failed jobs.

Puma plugin

We provide a Puma plugin if you want to run the Solid Queue's supervisor together with Puma and have Puma monitor and manage it. You just need to add

```
1 plugin :solid_queue
```

to your `puma.rb` configuration.

Jobs and transactional integrity

:warning: Having your jobs in the same ACID-compliant database as your application data enables a powerful yet sharp tool: taking advantage of transactional integrity to ensure some action in your app is not committed unless your job is also committed. This can be very powerful and useful, but it can also backfire if you base some of your logic on this behaviour, and in the future, you move to another active job backend, or if you simply move Solid Queue to its own database, and suddenly the behaviour changes under you.

By default, Solid Queue runs in the same DB as your app, and job enqueueing is *not* deferred until any ongoing transaction is committed, which means that by default, you'll be taking advantage of this transactional integrity.

If you prefer not to rely on this, or avoid relying on it unintentionally, you should make sure that: - You set `config.active_job.enqueue_after_transaction_commit` to `always`, if you're using Rails 7.2+. - Or, your jobs relying on specific records are always enqueued on `after_commit` callbacks or otherwise from a place where you're certain that whatever data the job will use has been committed to the database before the job is enqueued. - Or, you configure a database for Solid Queue, even if it's the same as your app, ensuring that a different connection on the thread handling requests or running jobs for your app will be used to enqueue jobs. For example:

```
1 class ApplicationRecord < ActiveRecord::Base
2   self.abstract_class = true
3
4   connects_to database: { writing: :primary, reading: :replica }
```

```
1 config.solid_queue.connects_to = { database: { writing: :primary,
2   reading: :replica } }
```

Recurring tasks

Solid Queue supports defining recurring tasks that run at specific times in the future, on a regular basis like cron jobs. These are managed by dispatcher processes and as such, they can be defined in the dispatcher's configuration like this:

```
1 dispatchers:
2   - polling_interval: 1
3     batch_size: 500
4     recurring_tasks:
5       my_periodic_job:
6         class: MyJob
7         args: [ 42, { status: "custom_status" } ]
8         schedule: every second
```

`recurring_tasks` is a hash/dictionary, and the key will be the task key internally. Each task needs to have a class, which will be the job class to enqueue, and a schedule. The schedule is parsed using Fugit, so it accepts anything that Fugit accepts as a cron. You can also provide arguments to be passed to the job, as a single argument, a hash, or an array of arguments that can also include kwargs as the last element in the array.

The job in the example configuration above will be enqueued every second as:

```
1 MyJob.perform_later(42, status: "custom_status")
```

Tasks are enqueued at their corresponding times by the dispatcher that owns them, and each task schedules the next one. This is pretty much inspired by what GoodJob does.

It's possible to run multiple dispatchers with the same `recurring_tasks` configuration. To avoid enqueueing duplicate tasks at the same time, an entry in a new `solid_queue_recurring_executions` table is created in the same transaction as the job is enqueued. This table has a unique index on `task_key` and `run_at`, ensuring only one entry per task per time will be created. This only works if you have `preserve_finished_jobs` set to `true` (the default), and the guarantee applies as long as you keep the jobs around.

Finally, it's possible to configure jobs that aren't handled by Solid Queue. That's it, you can have a job like this in your app:

```
1 class MyResqueJob < ApplicationJob
2   self.queue_adapter = :resque
3
4   def perform(arg)
5     # ..
6   end
7 end
```

You can still configure this in Solid Queue:

```
1  dispatchers:  
2    - recurring_tasks:  
3      my_periodic_resque_job:  
4        class: MyResqueJob  
5        args: 22  
6        schedule: "*/5 * * * *"
```

and the job will be enqueued via `perform_later` so it'll run in Resque. However, in this case we won't track any `solid_queue_recurring_execution` record for it and there won't be any guarantees that the job is enqueued only once each time.

Inspiration

Solid Queue has been inspired by resque and GoodJob. We recommend checking out these projects as they're great examples from which we've learnt a lot.

License

The gem is available as open source under the terms of the MIT License.