DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.

Active Job Basics

This guide provides you with all you need to get started in creating, enqueuing and executing background jobs.

After reading this guide, you will know:

- How to create jobs.
- How to enqueue jobs.
- How to run jobs in the background.
- How to send emails from your application asynchronously.

What is Active Job?

Active Job is a framework for declaring jobs and making them run on a variety of queuing backends. These jobs can be everything from regularly scheduled clean-ups, to billing charges, to mailings. Anything that can be chopped up into small units of work and run in parallel, really.

The Purpose of Active Job

The main point is to ensure that all Rails apps will have a job infrastructure in place. We can then have framework features and other gems build on top of that, without having to worry about API differences between various job runners such as Delayed Job and Resque. Picking your queuing backend becomes more of an operational concern, then. And you'll be able to switch between them without having to rewrite your jobs.

NOTE: Rails by default comes with an asynchronous queuing implementation that runs jobs with an in-process thread pool. Jobs will run asynchronously, but any jobs in the queue will be dropped upon restart.

Creating a Job

This section will provide a step-by-step guide to creating a job and enqueuing it.

Create the Job

Active Job provides a Rails generator to create jobs. The following will create a job in `app/jobs` (with an attached test case under `test/jobs`):

```
$ bin/rails generate job guests_cleanup
invoke  test_unit
```

```
create    test/jobs/guests_cleanup_job_test.rb
create  app/jobs/guests_cleanup_job.rb
```

You can also create a job that will run on a specific queue:

```
$ bin/rails generate job guests_cleanup --queue urgent
```

If you don't want to use a generator, you could create your own file inside of `app/jobs`, just make sure that it inherits from `ApplicationJob`.

Here's what a job looks like:

```
class GuestsCleanupJob < ApplicationJob
  queue_as :default

  def perform(*guests)
    # Do something later
  end
end
```

Note that you can define `perform` with as many arguments as you want.

Enqueue the Job

Enqueue a job using `perform_later` and, optionally, `set`. Like so:

```
# Enqueue a job to be performed as soon as the queuing system is
# free.
GuestsCleanupJob.perform_later guest

# Enqueue a job to be performed tomorrow at noon.
GuestsCleanupJob.set(wait_until: Date.tomorrow.noon).perform_later(guest)

# Enqueue a job to be performed 1 week from now.
GuestsCleanupJob.set(wait: 1.week).perform_later(guest)

# `perform_now` and `perform_later` will call `perform` under the hood so
# you can pass as many arguments as defined in the latter.
GuestsCleanupJob.perform_later(guest1, guest2, filter: 'some_filter')
```

That's it!

Job Execution

For enqueuing and executing jobs in production you need to set up a queuing backend, that is to say, you need to decide on a 3rd-party queuing library that Rails should use. Rails itself only provides an in-process queuing system, which only keeps the jobs in RAM. If the process crashes or the machine is reset, then all outstanding jobs are lost with the default async backend. This may be fine for smaller apps or non-critical jobs, but most production apps will need to pick a persistent backend.

Backends

Active Job has built-in adapters for multiple queuing backends (Sidekiq, Resque, Delayed Job, and others). To get an up-to-date list of the adapters see the API Documentation for `ActiveJob::QueueAdapters`.

Setting the Backend

You can easily set your queuing backend with `config.active_job.queue_adapter`:

```
# config/application.rb
module YourApp
  class Application < Rails::Application
    # Be sure to have the adapter's gem in your Gemfile
    # and follow the adapter's specific installation
    # and deployment instructions.
    config.active_job.queue_adapter = :sidekiq
  end
end
```

You can also configure your backend on a per job basis:

```
class GuestsCleanupJob < ApplicationJob
  self.queue_adapter = :resque
  # ...
end

# Now your job will use `resque` as its backend queue adapter, overriding what
# was configured in `config.active_job.queue_adapter`.
```

Starting the Backend

Since jobs run in parallel to your Rails application, most queuing libraries require that you start a library-specific queuing service (in addition to starting your Rails app) for the job processing to work. Refer to library documentation for instructions on starting your queue backend.

Here is a noncomprehensive list of documentation:

- Sidekiq
- Resque
- Sneakers
- Sucker Punch
- Queue Classic
- Delayed Job
- Que
- Good Job

Queues

Most of the adapters support multiple queues. With Active Job you can schedule the job to run on a specific queue using `queue_as`:

```
class GuestsCleanupJob < ApplicationJob
  queue_as :low_priority
  # ...
end
```

You can prefix the queue name for all your jobs using `config.active_job.queue_name_prefix` in `application.rb`:

```
# config/application.rb
module YourApp
  class Application < Rails::Application
    config.active_job.queue_name_prefix = Rails.env
  end
end
```

```
# app/jobs/guests_cleanup_job.rb
class GuestsCleanupJob < ApplicationJob
  queue_as :low_priority
  # ...
end

# Now your job will run on queue production_low_priority on your
# production environment and on staging_low_priority
# on your staging environment
```

You can also configure the prefix on a per job basis.

```
class GuestsCleanupJob < ApplicationJob
  queue_as :low_priority
  self.queue_name_prefix = nil
  # ...
end

# Now your job's queue won't be prefixed, overriding what
# was configured in `config.active_job.queue_name_prefix`.
```

The default queue name prefix delimiter is '_'. This can be changed by setting `config.active_job.queue_name_delimiter` in `application.rb`:

```
# config/application.rb
module YourApp
  class Application < Rails::Application
    config.active_job.queue_name_prefix = Rails.env
    config.active_job.queue_name_delimiter = '.'
```

```
  end
end

# app/jobs/guests_cleanup_job.rb
class GuestsCleanupJob < ApplicationJob
  queue_as :low_priority
  # ...
end

# Now your job will run on queue production.low_priority on your
# production environment and on staging.low_priority
# on your staging environment
```

If you want more control on what queue a job will be run you can pass a `:queue` option to `set`:

```
MyJob.set(queue: :another_queue).perform_later(record)
```

To control the queue from the job level you can pass a block to `queue_as`. The block will be executed in the job context (so it can access `self.arguments`), and it must return the queue name:

```
class ProcessVideoJob < ApplicationJob
  queue_as do
    video = self.arguments.first
    if video.owner.premium?
      :premium_videojobs
    else
      :videojobs
    end
  end

  def perform(video)
    # Do process video
  end
end

ProcessVideoJob.perform_later(Video.last)
```

NOTE: Make sure your queuing backend "listens" on your queue name. For some backends you need to specify the queues to listen to.

Callbacks

Active Job provides hooks to trigger logic during the life cycle of a job. Like other callbacks in Rails, you can implement the callbacks as ordinary methods and use a macro-style class method to register them as callbacks:

```
class GuestsCleanupJob < ApplicationJob
  queue_as :default
```

```ruby
  around_perform :around_cleanup

  def perform
    # Do something later
  end

  private
    def around_cleanup
      # Do something before perform
      yield
      # Do something after perform
    end
end
```

The macro-style class methods can also receive a block. Consider using this style if the code inside your block is so short that it fits in a single line. For example, you could send metrics for every job enqueued:

```ruby
class ApplicationJob < ActiveJob::Base
  before_enqueue { |job| $statsd.increment "#{job.class.name.underscore}.enqueue" }
end
```

Available callbacks

- before_enqueue
- around_enqueue
- after_enqueue
- before_perform
- around_perform
- after_perform

Action Mailer

One of the most common jobs in a modern web application is sending emails outside of the request-response cycle, so the user doesn't have to wait on it. Active Job is integrated with Action Mailer so you can easily send emails asynchronously:

```ruby
# If you want to send the email now use #deliver_now
UserMailer.welcome(@user).deliver_now

# If you want to send the email through Active Job use #deliver_later
UserMailer.welcome(@user).deliver_later
```

NOTE: Using the asynchronous queue from a Rake task (for example, to send an email using `.deliver_later`) will generally not work because Rake will likely end, causing the in-process thread pool to be deleted, before any/all of the

`.deliver_later` emails are processed. To avoid this problem, use `.deliver_now` or run a persistent queue in development.

Internationalization

Each job uses the `I18n.locale` set when the job was created. This is useful if you send emails asynchronously:

```
I18n.locale = :eo

UserMailer.welcome(@user).deliver_later # Email will be localized to Esperanto.
```

Supported types for arguments

ActiveJob supports the following types of arguments by default:

- Basic types (`NilClass`, `String`, `Integer`, `Float`, `BigDecimal`, `TrueClass`, `FalseClass`)
- `Symbol`
- `Date`
- `Time`
- `DateTime`
- `ActiveSupport::TimeWithZone`
- `ActiveSupport::Duration`
- `Hash` (Keys should be of `String` or `Symbol` type)
- `ActiveSupport::HashWithIndifferentAccess`
- `Array`
- `Range`
- `Module`
- `Class`

GlobalID

Active Job supports GlobalID for parameters. This makes it possible to pass live Active Record objects to your job instead of class/id pairs, which you then have to manually deserialize. Before, jobs would look like this:

```
class TrashableCleanupJob < ApplicationJob
  def perform(trashable_class, trashable_id, depth)
    trashable = trashable_class.constantize.find(trashable_id)
    trashable.cleanup(depth)
  end
end
```

Now you can simply do:

```
class TrashableCleanupJob < ApplicationJob
  def perform(trashable, depth)
    trashable.cleanup(depth)
```

```
  end
end
```

This works with any class that mixes in `GlobalID::Identification`, which by default has been mixed into Active Record classes.

Serializers

You can extend the list of supported argument types. You just need to define your own serializer:

```
# app/serializers/money_serializer.rb
class MoneySerializer < ActiveJob::Serializers::ObjectSerializer
  # Checks if an argument should be serialized by this serializer.
  def serialize?(argument)
    argument.is_a? Money
  end

  # Converts an object to a simpler representative using supported object types.
  # The recommended representative is a Hash with a specific key. Keys can be of basic types
  # You should call `super` to add the custom serializer type to the hash.
  def serialize(money)
    super(
      "amount" => money.amount,
      "currency" => money.currency
    )
  end

  # Converts serialized value into a proper object.
  def deserialize(hash)
    Money.new(hash["amount"], hash["currency"])
  end
end
```

and add this serializer to the list:

```
# config/initializers/custom_serializers.rb
Rails.application.config.active_job.custom_serializers << MoneySerializer
```

Note that autoloading reloadable code during initialization is not supported. Thus it is recommended to set-up serializers to be loaded only once, e.g. by amending `config/application.rb` like this:

```
# config/application.rb
module YourApp
  class Application < Rails::Application
    config.autoload_once_paths << Rails.root.join('app', 'serializers')
  end
end
```

Exceptions

Exceptions raised during the execution of the job can be handled with `rescue_from`:

```
class GuestsCleanupJob < ApplicationJob
  queue_as :default

  rescue_from(ActiveRecord::RecordNotFound) do |exception|
    # Do something with the exception
  end

  def perform
    # Do something later
  end
end
```

If an exception from a job is not rescued, then the job is referred to as "failed".

Retrying or Discarding failed jobs

A failed job will not be retried, unless configured otherwise.

It's possible to retry or discard a failed job by using `retry_on` or `discard_on`, respectively. For example:

```
class RemoteServiceJob < ApplicationJob
  retry_on CustomAppException # defaults to 3s wait, 5 attempts

  discard_on ActiveJob::DeserializationError

  def perform(*args)
    # Might raise CustomAppException or ActiveJob::DeserializationError
  end
end
```

Deserialization

GlobalID allows serializing full Active Record objects passed to `#perform`.

If a passed record is deleted after the job is enqueued but before the `#perform` method is called Active Job will raise an `ActiveJob::DeserializationError` exception.

Job Testing

You can find detailed instructions on how to test your jobs in the testing guide.