Active Job Basics

January 13, 2015

This guide provides you with all you need to get started in creating, enqueueing 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 async.

1 Introduction

Active Job is a framework for declaring jobs and making them run on a variety of queueing 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.

2 The Purpose of Active Job

The main point is to ensure that all Rails apps will have a job infrastructure in place, even if it's in the form of an "immediate runner". 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.

3 Creating a Job

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

3.1 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 `ActiveJob::Base`.

Here's what a job looks like:

```
class GuestsCleanupJob < ActiveJob::Base
  queue_as :default

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

3.2 Enqueue the Job

Enqueue a job like so:

```
# Enqueue a job to be performed as soon the queueing system is free.
MyJob.perform_later record

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

# Enqueue a job to be performed 1 week from now.
MyJob.set(wait: 1.week).perform_later(record)
```

That's it!

4 Job Execution

If no adapter is set, the job is immediately executed.

4.1 Backends

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

4.2 Setting the Backend

You can easily set your queueing backend:

```
# 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
```

5 Queues

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

```
class GuestsCleanupJob < ActiveJob::Base
  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.rb
class GuestsCleanupJob < ActiveJob::Base
  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
```

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.rb
class GuestsCleanupJob < ActiveJob::Base
  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 you can access `self.arguments`) and you must return the queue name:

```
class ProcessVideoJob < ActiveJob::Base
  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)
```

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

6 Callbacks

Active Job provides hooks during the lifecycle of a job. Callbacks allow you to trigger logic during the lifecycle of a job.

6.1 Available callbacks

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

6.2 Usage

```
class GuestsCleanupJob < ActiveJob::Base
  queue_as :default

  before_enqueue do |job|
    # do something with the job instance
  end

  around_perform do |job, block|
    # do something before perform
    block.call
    # do something after perform
  end

  def perform
    # Do something later
  end
end
```

7 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:

```
# 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
```

8 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 < ActiveJob::Base
  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 < ActiveJob::Base
  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 Model classes.

9 Exceptions

Active Job provides a way to catch exceptions raised during the execution of the job:

```
class GuestsCleanupJob < ActiveJob::Base
  queue_as :default

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

  def perform
    # Do something later
  end
end
```

10 Feedback

You're encouraged to help improve the quality of this guide.

Please contribute if you see any typos or factual errors. To get started, you can read our documentation contributions section.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Make sure to check Edge Guides first to verify if the issues are already fixed or not on the master branch. Check the Ruby on Rails Guides Guidelines for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please open an issue.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the rubyonrails-docs mailing list.
