Inside ActiveJob

Jerry D'Antonio
@jerrydantonio
jerrydantonio.com

About Me

- I live and work in Akron, OH
- I work at Test Double
    - Mission: Improve the way the world builds software
- I am the creator and lead maintainer of Concurrent Ruby
    - Used by Rails, Sidekiq, Elasticsearch, MS Azure, Sucker Punch, others
    - http://concurrent-ruby.com

Assumptions

- You know what ActiveJob is
- You understand what tasks are best suited for asynchronous job processing
- You have used ActiveJob in a production Rails application
- You are familiar with at least one supported job processor
- You have a basic understanding of concurrency and parallelism

What is ActiveJob?

“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.”

--http://guides.rubyonrails.org/active_job_basics.html

- Framework: It unifies pre-existing and subsequently-created job runners
    - Supports Backburner, Delayed Job, Qu, Que, Resque, Sidekiq, Sneakers, Sucker Punch
- Scheduled: Supports ASAP and time-based scheduling
- Parallel: Can potentially scale across processors and machines

Why do we need ActiveJob?

- Background job processors exist because they solve a problem
    - Long-running tasks block your HTTP response
    - Not every operation must occur before you send a response
    - Some tasks can be performed after the response so long as there is a guarantee
    - Background job processors provide the scheduling and guarantees

Why do we need ActiveJob?

- ActiveJob unifies the ecosystem of job processors
    - Job processors existed but each was unique
    - Switching from one to another often required significant refactoring
    - ActiveJob provides an abstraction layer which supports the most common features
    - “Picking your queuing backend becomes more of an operational concern.”
      - Rails Guides

A Simple Job Class

```
class DoSomethingLaterJob < ActiveJob::Base

  queue_as :default # optional

  def perform(*args)

    # Do something later

  end

end
```

Using Our Simple Job Class

```
# config/application.rb

module YourApp
  class Application < Rails::Application
    config.active_job.queue_adapter = :inside_job
  end
end

# later, probably in a controller

DoSomethingLaterJob.perform_later(user)

DoSomethingLaterJob.set(wait_until: Date.tomorrow.noon).perform_later(user)
```

Building Our Async Backend

- Threaded or forked?
  - We’re going to use concurrent-ruby thread pools
- Persisting our jobs
  - For simplicity, we’ll store our job data in-memory (no database persistence)
  - Our job processor will only be suitable for development and testing

Building Our Async Backend

- The pieces
  - ActiveJob::Core—the job metadata
  - Queue adapter—marshals the job between Rails and the job runner
  - Job runner—provides asynchronous behavior
- The job runner is independent of Rails (Sidekiq, Sucker Punch, etc.)
- The queue adapter is in the Rails ActiveJob gem

ActiveJob::Core Class

- Is the object passed into the queue adapter when a job is enqueued
- Provides two hugely important methods:
  - `#serialize`
  - `#deserialize`

ActiveJob::Core Class

- Has several useful attributes:
  - `:queue_name`
  - `:priority`
  - `:scheduled_at`
  - `:job_id`
  - `:provider_job_id`

Our Queue Adapter

```
class InsideJobAdapter
  def enqueue(job)
    InsideJob.enqueue(job.serialize, queue: job.queue_name)
  end
  def enqueue_at(job, timestamp)
    InsideJob.enqueue_at(job.serialize, timestamp, queue: job.queue_name)
  end
end
```

Our Job Runner: The Thread Pool

- What we need
  - Jobs are post to queues
  - Jobs must run asynchronously
- What we have
  - Thread pools from Concurrent Ruby each have their own queue and one or more threads
  - So a new thread pool for each job queue is all we need

Our Job Runner: Creating Queues

```
QUEUES = Concurrent::Map.new do |hash, queue_name|
  hash.compute_if_absent(queue_name) do
    InsideJob.create_thread_pool
  end
end
…
def create_thread_pool
  Concurrent::CachedThreadPool.new
end
```

Our Job Runner: Enqueue Now

```
def enqueue(job_data, queue: 'default')
  QUEUES[queue].post(job_data) do |job|
    ActiveJob::Base.execute(job)
  end
end
```

Our Job Runner: Enqueue For Later

```
def enqueue_at(job_data, timestamp, queue: 'default')
  delay = timestamp - Time.current.to_f

  if delay > 0
    Concurrent::ScheduledTask.execute(
      delay, args: [job_data], executor: QUEUES[queue]) do |job|

      ActiveJob::Base.execute(job)
    end
  else
    enqueue(job_data, queue: queue)
  end
end
```

```
class InsideJob

  QUEUES = ThreadSafe::Cache.new do |hash, queue_name|
    hash.compute_if_absent(queue_name) { ActiveJob::InsideJob.create_thread_pool }
  end

  class << self
    def create_thread_pool
      Concurrent::CachedThreadPool.new
    end

    def enqueue(job_data, queue: 'default')
      QUEUES[queue].post(job_data) { |job| ActiveJob::Base.execute(job) }
    end

    def enqueue_at(job_data, timestamp, queue: 'default')
      delay = timestamp - Time.current.to_f
      if delay > 0
        Concurrent::ScheduledTask.execute(delay, args: [job_data], executor: QUEUES[queue]) do |job|
          ActiveJob::Base.execute(job)
        end
      else
        enqueue(job_data, queue: queue)
      end
    end
  end
end
```

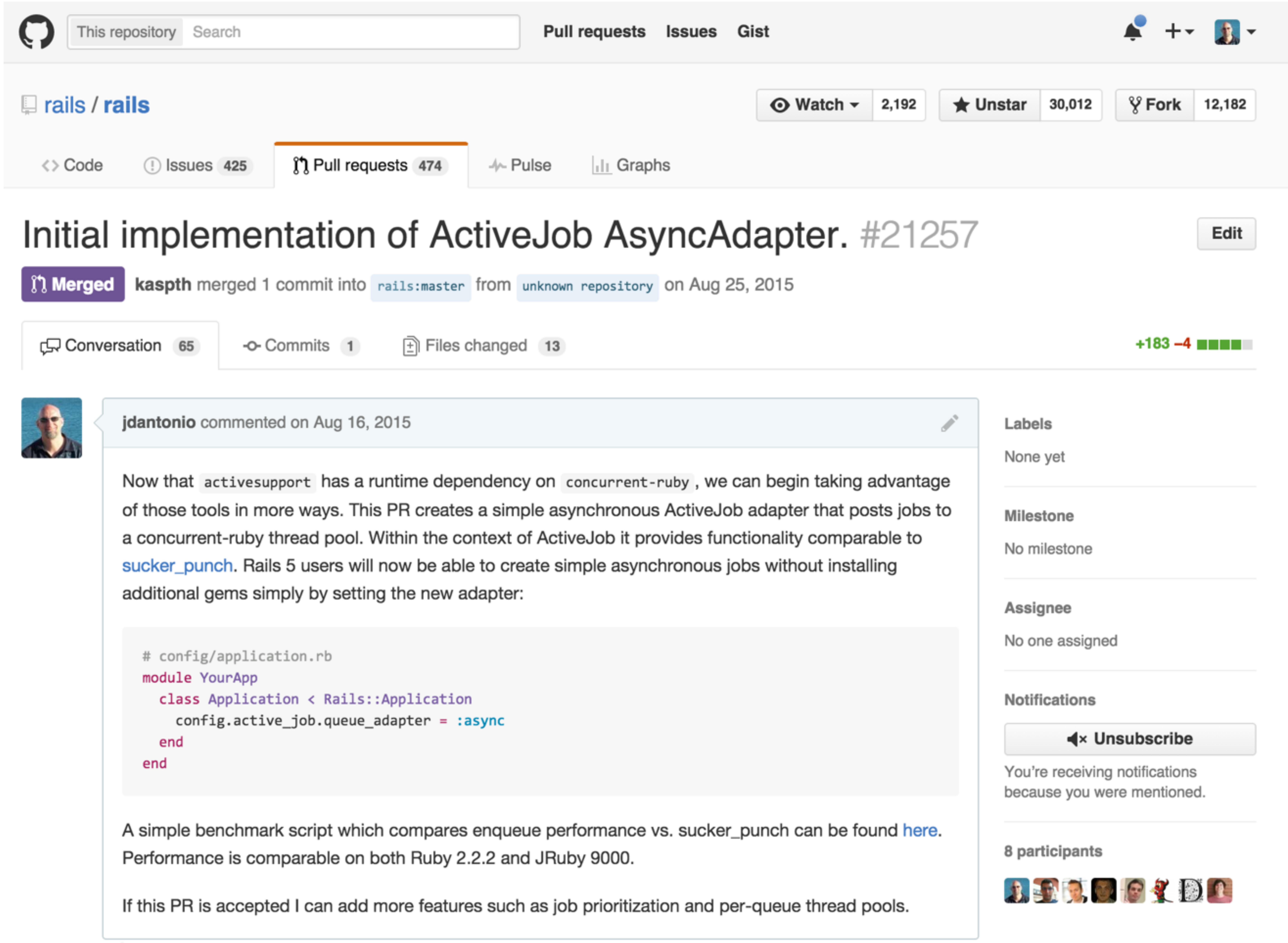
```ruby
module ActiveJob
  module QueueAdapters

    class InsideJobAdapter

      def enqueue(job)
        ActiveJob::InsideJob.enqueue(job.serialize, queue: job.queue_name)
      end

      def enqueue_at(job, timestamp)
        ActiveJob::InsideJob.enqueue_at(job.serialize, timestamp, queue: job.queue_name)
      end
    end
  end
end
```

Rails AsyncJob


This repository Search
Pull requests Issues Gist
rails / rails
Watch 2,192 Unstar 30,012 Fork 12,182
Code Issues 425 Pull requests 474 Pulse Graphs
Edit

Initial implementation of ActiveJob AsyncAdapter. #21257

Merged **kaspth** merged 1 commit into `rails:master` from `unknown repository` on Aug 25, 2015

Conversation 65 Commits 1 Files changed 13
+183 −4

jdantonio commented on Aug 16, 2015

Now that `activesupport` has a runtime dependency on `concurrent-ruby`, we can begin taking advantage of those tools in more ways. This PR creates a simple asynchronous ActiveJob adapter that posts jobs to a concurrent-ruby thread pool. Within the context of ActiveJob it provides functionality comparable to sucker_punch. Rails 5 users will now be able to create simple asynchronous jobs without installing additional gems simply by setting the new adapter:

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

A simple benchmark script which compares enqueue performance vs. sucker_punch can be found here. Performance is comparable on both Ruby 2.2.2 and JRuby 9000.

If this PR is accepted I can add more features such as job prioritization and per-queue thread pools.

Subsequent AsyncJob Refactoring

- Collapse into one file
- Rename stuff
- Assign a provider job id
- Use one thread pool for everything
    - Throttle the thread pool
    - Configure the thread pool

Going Further

- Sucker Punch
  - Threaded, in-memory, asynchronous job processor
  - Uses Concurrent Ruby thread pools
  - Decorates every job to provide job statistics and greater job control
  - Includes advanced shutdown behavior and options
- Sidekiq
  - Threaded job processor using database persistence
  - Includes many advanced features

My name is Jerry D'Antonio

Please tweet me @jerrydantonio &

Say hello@testdouble.com