

td

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>



Mission: Our mission is ambitious, but we believe it is something that every developer is capable of. You just have to try.

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

Track: Behind the Magic

Intended to be a deep dive.

We will actually be building a real (but limited) asynchronous job processor.

In the interest of time we must skip the introductory material.

If you would like an introduction to Active Job, please consult the Rails Guides. They are excellent.

If you would an introduction to concurrency, please check out my RubyConf talk “Everything You Know About the GIL is Wrong”—available on YouTube.

<https://www.youtube.com/watch?v=dP4U1yl1WZ0>

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

NOTE: Pre-existing backends

* ActiveJob wasn't introduced until Rails 4

* Asynchronous job processors already existed

* There was a need and the community filled it

* ActiveJob was not intended to replace these existing queuing systems

* It was intended to unify them under a single API

* Much the was ActiveRecord unifies relational databases

* Also like ActiveJob, not all features of all job processors can be accessed via ActiveJob

* So ActiveJob needed to work with these existing systems

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
```

Basic job class
* Extend ActiveSupport
* Optionally set the queue
* Define a #perform method that does something useful

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)
```

Our job processor
* Will be called InsideJob
* So we configure the adapter using the symbol version
* Standard Rails inflections apply

Posting a job
* Use the #perform_later method to enqueue an ASAP job
* Use the #set method to enqueue a scheduled task

NOTE: These are class methods—these are stateless functions
* We'll talk about this more later

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

Forked

- * Spawns one or more separate processes for the async tasks
- * Full parallelization across multiple processors or cores
- * Requires a completely separate application for job processing
- * Also requires separate processes monitoring tools
- * Individual workers can crash without affecting the Rails application

Threaded

- * Runs the task in the same process as Rails
- * Constrained to one processor/core and limited by the GIL on MRI
 - * But fully parallelized on JRuby and Rubinius
- * Crashed threads must be monitored from within the application

Persistence

- * When job data is stored in-memory only, unprocessed jobs will not survive shutdown of the Rails app
- * Job data persisted to a data store will still be around after a restart of Rails
- * Job persistence involves significantly more latency

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

* Queue adapters are a part of Rails and are maintained by the Rails core team

* But processor maintainers are ultimately responsible for keeping the associated adapter up to date

* Rails provides a suite of unit tests for the queue supported queue adapters

* This ensures that all supported processors meet the minimum specified behavior

* But most job processors can be used independently of ActiveJob, often providing additional capabilities

* Rails Guides documents which processors are supported and which features they support

* If a new job processor were created it could potentially be added to Rails via adapter PR

ActiveJob::Core Class

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

Before we talk about the queue adapter and the job processor, let's talk about ActiveJob::Core

- * The Core class represents an individual job which has been post by the Rails application
- * The adapter is responsible for ensuing the job with the processor
- * The processor is responsible for actually running the job, and all that entails (threads/processes, error handling, etc.)
- * The job class is the glue that binds the two
 - * It includes job metadata and the actual job proc

I'll talk about job serialization more later

ActiveJob::Core Class

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

Job metadata

- * :queue_name is the name of the queue to which the job is to be post
 - * When no queue name is provided the job should go to the default queue
- * :priority specifies the relative importance of the given job with respect to the other jobs post to the queue
 - * Higher priority jobs will be processed before lower priority jobs
 - * Not all job processors support priority
- * :scheduled_at is used when the job is intended to be run at the specified time
 - * Without this field the job is ASAP and will be run as soon as the processor can grab it off the queue
- * :job_id is a unique ID assigned by Rails
 - * Should generally not be used by the processor because it cab only guarantee uniqueness within a single Rails instance
- * :provider_job_id a unique ID assigned by the job processor; guaranteed to be unique within the scope of the processor

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
```

- Starting with the adapter is an outside-in approach
- * Psuedo-TDD; we're specifying the processor API first
 - * The adapter simply marshals the job to the processor
 - * Since we are building a new processor (not adapting to a pre-existing one) we can define a very simple API

Note the use of class methods on InsideJob

- * This is, obviously, not very OO
- * But it reinforces the idea of statelessness expressed earlier
- * We are passing a job off to a queue to be processed later by another thread/process
- * ALL of the state is encapsulated in the Job object
- * Maintaining state across calls is inherently NOT thread safe
 - * Rails may be running on a multithreaded web server
 - * Which means these calls may be coming in from multiple threads
 - * The job itself WILL be processed on another thread or completely different process
 - * So it is essential that we be entirely stateless

Note that we serialize the job

- * Strictly speaking, passing the job object is not thread-safe
 - * The job object may contain references to parameters
 - * It is possible that those objects are still referenced by other code
 - * Though unlikely (given the usage pattern), it is possible that we're passing unsafe references
 - * Serialization makes copies that can be safely passed
- * Serialization is more important when persisting the job to a data store
 - * Most databases and data stores cannot store real Ruby objects
 - * One option is to marshall to a binary object
 - * A better option is to serialize the object as JSON
 - * JSON is just text, so all datastore can handle it
 - * Many modern data stores can handle JSON directly and very efficiently
 - * Using JSON also encourages better design—passing simple values rather than complex objects
- * Serialization by Rails itself allows us to switch job processors more easily
 - * Every job processor uses the same serialization so we don't have to worry about compatibility
 - * Which is especially important when we use different processors in development, test, and production
- * We serialize because
 - * It's thread safe
 - * We want the best possible interoperability

This class is very simple, because it CAN be

- * All it needs to do is marshal the job metadata between Rails and the job processor
- * Since we are creating a new job processor we have no need for additional complexity
- * The queue adapters for the pre-existing processors may be a little more complex, but not by much

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

Concurrent Ruby cached thread pool is perfectly suited for posting jobs

- * It dynamically adds threads as needed
- * It prunes idle threads
- * And it has no maximum queue size

PSA: Please do NOT use thread pools directly in your code

- * Thread pools are a low-level abstraction intended to be used internally within Concurrent Ruby
- * Concurrent Ruby lazily initializes and actively manages a pair of thread pools for you
- * All of our high-level abstractions use these thread pools
 - * Future, Promise, Actor, Agent, ScheduledTask, TimerTask, the Async mixin, etc.
 - * They also provide robust, consistent, idiomatic ways of handling errors, return values, etc.
 - * Most Rubyists should only use the high-level abstractions
- * In this case we're using a thread pool directly because we have very specific, low-level needs
- * This is not the average use case

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
```

Automatically creating new thread pools

- * Each "queue" will be represented by a single thread pool
 - * A thread-safe map (hash) class will be used to map queue names to thread pools
- * But we don't know in advance what queues we will have
 - * The ActiveSupport API allows us to specify the queue name in each job class
 - * So we need to dynamically define queues/thread pools whenever we encounter a new queue name
- * Much like a Ruby Hash, Map's initializer can be given a block which will be processed when a key isn't found
 - * Because this is a multi-threaded system we need to be thread-safe
 - * Map's #compute_if_absent method is atomic and synchronized
 - * So we can safely call our queue/thread pool creation method in the block
 - * NOTE: This looks odd because thread-safety is hard; Map is a very low-level abstraction

Our Job Runner: Enqueue Now

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

When we enqueue a job we simply post it to the appropriate queue/thread pool

- * This is thread-safe since we're using a Concurrent::Map
- * And we don't have to worry about an undefined queue—we handled that in the Map's initializer
- * So we just get the appropriate thread pool and post the job
- * The block passed to #post is what will be run when the thread pool dequeues the task
- * Unsurprisingly, the #post method is thread-safe

Executing the actual job

- * Within the thread pool's task we call ActiveJob::Base.execute
 - * We must pass the job object itself as the only parameter
- * ActiveJob::Base will handle all the details
 - * It will interrogate the job object to figure out which job class it belongs to
 - * It will then create a new instance of that class
 - * And it will call the #perform method, passing the original arguments (stored within the job object)

ActiveJob::Base does all the heavy lifting

Our job is simply to provide the marshaling and concurrency

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|

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

Enqueuing a job for later is a little more complicated

- * Scheduled jobs need to happen at a later time
- * ActiveSupport provides a rich API for specifying when jobs happen
- * But ActiveSupport also is kind enough to convert that time value into a simple, integer delay in seconds

Concurrent Ruby's ScheduledTask to the rescue!

- * A scheduled task is exactly what it implies—it's an asynchronous task that occurs at a scheduled time
- * By default, ScheduledTask runs on the global thread pool, but the API supports dependency injection
 - * The :executor option exists on most Concurrent Ruby high-level abstractions
 - * This allows developers with advanced needs to get the benefits of the high-level abstractions while also configuring their own thread pools
 - * Executor injection is idiomatic—attempting to manually monkey with the global thread pools will make Jerry a sad panda
- * So we simply create a new scheduled task and pass it the appropriate queue and job
- * When it executes it simply passes the call to ActiveSupport::Base

Although not strictly necessary, we're going to check the delay and short-circuit if we can

```

class InsideJob

  QUEUES = ThreadSafe::Cache.new do |hash, queue_name|
    hash.compute_if_absent(queue_name) { ActiveSupport::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| ActiveSupport::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|
          ActiveSupport::Base.execute(job)
        end
      else
        enqueue(job_data, queue: queue)
      end
    end
  end
end
end

```

This is the job processor class.

This is way too much code for one slide but I am showing it for one reason:

- * To show how simple it is to create a minimal job processor given the right tools
- * All we need to do is provide the asynchronous behavior.
- * ActiveSupport does the rest.

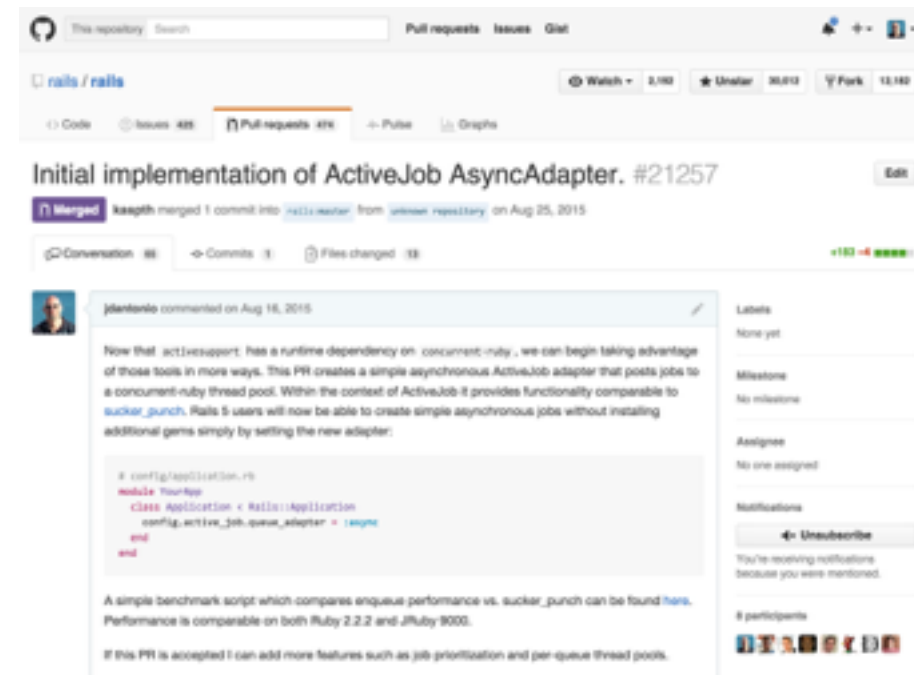
```
module ActiveJob
  module QueueAdapters

    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



The screenshot shows a GitHub pull request titled "Initial implementation of ActiveJob AsyncAdapter. #21257" in the rails/rails repository. The pull request is merged and was created on August 25, 2015. It includes a comment from jteniente dated August 18, 2015, which explains the motivation for the change: to take advantage of the concurrent-ruby dependency in ActiveSupport. The comment includes a code snippet for config/application.rb showing the configuration for the :async adapter. It also mentions a benchmark script comparing enqueue performance with sucker_punch and provides a link to it. The right sidebar shows the pull request's status, including labels, milestones, assignees, and notifications.

Initial implementation of ActiveJob AsyncAdapter. #21257

merged kaseph merged 1 commit into rails/master from jteniente:rails_async_adapter on Aug 25, 2015

Conversation 88 Commits 3 Files changed 18 +183 -4

jteniente commented on Aug 18, 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.

Labels: None yet

Milestone: No milestone

Assignee: No one assigned

Notifications: You're receiving notifications because you were mentioned.

8 participants

Where do I find the code? In Rails 5!

It's not InsideJob, it's AsyncJob

- * Rails already has an :inline adapter for development and testing
- * I suggested adding a similar :async adapter/processor
- * The Rails team liked the idea so we worked together on an implementation
- * This new processor is available in Rails 5
- * Simply set the queue_adapter to :async
- * It's also the default adapter. If you don't specify an adapter, this is what you'll get.

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

The current implementation has evolved and doesn't look exactly like what we saw here

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

Sidekiq

- * Does not use Concurrent Ruby thread pools
- * Creates and manages its own threads
- * Number of threads remains static while running —no dynamic thread creation
- * But uses Concurrent Ruby for low level locking and synchronization

The logo consists of the lowercase letters 'td' in a black, cursive script font. The 't' and 'd' are connected, with the 'd' having a small loop at the bottom. The logo is positioned in the top-left corner of a white rectangular box that has a thin black border and a bright green horizontal line at the top.

My name is Jerry D'Antonio

Please tweet me @jerrydantonio &

Say hello@testdouble.com