

# Background jobs with Rails

by Rino Kovačević

# What are background jobs?

**"Background jobs** can dramatically improve the scalability of a web app by enabling it to offload slow or CPU-intensive tasks from its front-end. This helps ensure that the front-end can handle incoming web requests promptly, reducing the likelihood of performance issues that occur when requests become backlogged." (from Heroku)

**Background jobs** are pieces of code that run **asynchronously** as background processes.



# Why use background jobs?

- Scheduling tasks for a certain time
  - Send an email at 10:00 pm
- For tasks we want to do periodically
  - Send weekly recap emails every Friday
- We want to do more things at the same time (asynchronously)
  - Send 1,000 emails now



# What can we use background jobs for?

- Emails / SMSs
- Notifications
- Data generation
- Billing processing
- Tasks that require third party services or APIs
- Better user experience



# How are they implemented in Rails?

## Active Job

- framework for declaring jobs and making them run on a variety of queuing backends (adapters)
- Ensure that all Rails apps have job infrastructure in place
- Active Job was first included in Rails 4.2 in 2014
- Single Standard Interface for already existing queues
- Backend for Action Mailer's **deliver\_later** functionality
- Comes with Generator



# Active Job adapters

- Most common used:
  - Delayed Job
  - Sidekiq
  - Resque

	Async	Queues	Delayed	Priorities	Timeout	Retries
Backburner	Yes	Yes	Yes	Yes	Job	Global
Delayed Job	Yes	Yes	Yes	Job	Global	Global
Que	Yes	Yes	Yes	Job	No	Job
queue_classic	Yes	Yes	Yes*	No	No	No
Resque	Yes	Yes	Yes (Gem)	Queue	Global	Yes
Sidekiq	Yes	Yes	Yes	Queue	No	Job
Sneakers	Yes	Yes	No	Queue	Queue	No
Sucker Punch	Yes	Yes	Yes	No	No	No
Active Job Async	Yes	Yes	Yes	No	No	No
Active Job Inline	No	Yes	N/A	N/A	N/A	N/A



# Create the Job

- Active Job provides a Rails generator

```
→ i386 devot_web git:(master) rails generate job registration_email
Running via Spring preloader in process 21109
  invoke  rspec
  create  spec/jobs/registration_email_job_spec.rb
  create  app/jobs/registration_email_job.rb
```

```
1  class RegistrationEmailJob < ApplicationJob
2    queue_as :default
3
4    def perform(*args)
5      # Do something later
6    end
7  end
```

```
1  require 'rails_helper'
2
3  RSpec.describe RegistrationEmailJob, type: :job do
4    pending "add some examples to (or delete) #{__FILE__}"
5  end
```



# Enqueue the Job

- Using `perform_later`
- Optionally set

```
24 def send_registration_email
25   # Enqueue a job to be performed as soon as the queuing system is
26   # free.
27   RegistrationEmailJob.perform_later guest_user
28
29   # Enqueue a job to be performed tomorrow at noon.
30   RegistrationEmailJob.set(wait_until: Date.tomorrow.noon).perform_later(guest_user)
31
32   # Enqueue a job to be performed 1 week from now.
33   RegistrationEmailJob.set(wait: 1.week).perform_later(guest_user)
34
35   # `perform_now` and `perform_later` will call `perform` under the hood so
36   # you can pass as many arguments as defined in the latter.
37   RegistrationEmailJob.perform_later(guest1, guest2, filter: 'some_filter')
38 end
```





# Execute the Job

- Set up a queuing backend
- Choose 3rd-party queuing library
  - by default, Rails only provides an in-process queuing system, which only keeps the jobs in RAM
- Example library - **Delayed job**



# Backend setup

- Add gem '**delayed\_job\_active\_record**' to your Gemfile
- Run:
  - Bundle install
  - Generate migration
  - Run migration

```
Installing delayed_job 4.1.11
Fetching delayed_job_active_record 4.1.7
Installing delayed_job_active_record 4.1.7
Bundle complete! 42 Gemfile dependencies, 138 gems now installed.
Use `bundle info [gemname]` to see where a bundled gem is installed.
→ i386 devot_web git:(master) x rails generate delayed_job:active_record
Running via Spring preloader in process 70223
    create  bin/delayed_job
    chmod  bin/delayed_job
    create  db/migrate/20230216111646_create_delayed_jobs.rb
→ i386 devot_web git:(master) x rake db:migrate
== 20230216111646 CreateDelayedJobs: migrating =====
-- create_table(:delayed_jobs)
   -> 0.0394s
-- add_index(:delayed_jobs, [:priority, :run_at], {:name=>"delayed_jobs_priority"})
   -> 0.0025s
== 20230216111646 CreateDelayedJobs: migrated (0.0420s) =====
```



# Backend setup

- Easy set queueing backend adapter
  - config/application.rb

```
9  module Devot
10  class Application < Rails::Application
11    # Initialize configuration defaults for originally generated Rails version.
12    config.load_defaults 6.1
13    config.assets.paths << Rails.root.join('app', 'assets', 'fonts')
14
15    # Configuration for the application, engines, and railties goes here.
16    #
17    # These settings can be overridden in specific environments using the files
18    # in config/environments, which are processed later.
19
20    config.active_job.queue_adapter = :delayed_job
21    config.autoload_paths += %W(#{config.root}/lib)
22    config.exceptions_app = routes
23  end
24 end
```



# Running Jobs

- Manually invoke rake jobs:work
  - Each instance of rake jobs:work represents one worker
  - Delete all jobs in queue with rake jobs:clear
- They can run on any computer/server
  - They periodically connect to the database



# Features

- Queues (e.g., queue\_as :low\_priority)
- Callbacks (after\_enqueue, before\_perform, after\_perform...)
- Action Mailer

```
24 UserMailer.welcome(@user).deliver_now
25 UserMailer.welcome(@user).deliver_later
```

- Exceptions

```
4 rescue_from(ActiveRecord::RecordNotFound) do |exception|
5   # Do something with the exception
6 end
```

- Retrying or discarding failed jobs

```
8 retry_on CustomAppException # defaults to 3s wait, 5 attempts
9
10 discard_on(CustomAppException) do |job, error|
11   ExceptionNotifier.caught(error)
12 end
```



# Testing jobs

- Test Jobs themselves
- Test that entities correctly enqueue Jobs

```
3 RSpec.describe RegistrationEmailJob, type: :job do
4   subject(:job) { described_class.perform_later(123) }
5
6   it 'queues the job' do
7     expect { job }
8       .to change(ActiveJob::Base.queue_adapter.enqueued_jobs, :size).by(1)
9   end
10
11   it 'is in default queue' do
12     expect(MyJob.new.queue_name).to eq('default')
13   end
14
15   it 'executes perform' do
16     expect(MyService).to receive(:call).with(123)
17     perform_enqueued_jobs { job }
18   end
19 end
```

```
19
20   it 'handles no results error' do
21     allow(MyService).to receive(:call).and_raise(NoResultsError)
22
23     perform_enqueued_jobs do
24       expect_any_instance_of(MyJob)
25         .to receive(:retry_job).with(wait: 10.minutes, queue: :default)
26
27       job
28     end
29   end
30
31   after do
32     clear_enqueued_jobs
33     clear_performed_jobs
34   end
35 end
```



## Story 1: Adapter changing

```
config.active_job.queue_adapter = :delayed_job
```

- We had problems with sending a lot of SMSs at the same time (~10,000) – we had 50 workers but it was still slow
- Our original implementation used Delayed Jobs – so we decided to try out Sidekiq and Resque
- We learnt how easy it is – and that they didn't effect our speed



## Story 2: Why are all adapters slow?

- The app itself would also become slower when the campaigns were running
- Database connections are limited
- When testing - everything was faster
- Database upgrade + code optimization solved our problems (and we went back to Delayed Job, because we like it's UI)





## Story 3: Now send 50,000 SMSs at the same time!

- Everything works great for 10,000 – why not for 50,000?
- We had adjust our code again – never have more than 1,000 Jobs in the queue with the same name ran at the same time
- Splitting the flow into smaller batches allowed us to track errors better – but also, magically solved our problems




# Common issues

- Job failures
- Memory leaks
- Deadlocks
- Slow performance
- Concurrency issues
- Resource contention
- Configuration issues



When working with background jobs in Rails, what are common errors that can occur?



When working with background jobs in Rails, some common errors that can occur include:  

1. Job failures: Jobs can fail for various reasons, such as network timeouts, invalid arguments, or resource limitations. To handle job failures, you can use error handling mechanisms, such as retrying failed jobs, logging errors, or sending notifications.
2. Memory leaks: Background jobs can consume a lot of memory if they are not designed properly. To prevent memory leaks, you should avoid creating unnecessary objects, use lazy loading, and free resources as soon as possible.
3. Deadlocks: Deadlocks occur when two or more processes are waiting for each other to release a resource. To avoid deadlocks, you should use locks sparingly, avoid nested locks, and release locks as soon as possible.
4. Slow performance: Background jobs can slow down your application if they take too long to complete or use too much CPU or memory. To improve performance, you should optimize your code, use efficient algorithms, and avoid unnecessary database queries.
5. Concurrency issues: If you have multiple background workers running at the same time, you may encounter concurrency issues, such as race conditions or inconsistent data. To avoid concurrency issues, you should use locking mechanisms, such as semaphores or mutexes, and ensure that your code is thread-safe.
6. Resource contention: Background jobs can compete for the same resources, such as database connections or file handles, which can cause resource contention. To reduce resource contention, you should use connection pooling, limit the number of concurrent workers, or use separate worker processes for different types of jobs.
7. Configuration issues: Background job frameworks, such as Sidekiq or Resque, require configuration settings, such as Redis or RabbitMQ connection details. If these settings are not configured correctly, jobs may fail to enqueue or run. To avoid configuration issues, you should review the documentation carefully and test your configuration thoroughly.



# Thank you for your time!

Questions?